Python Cheat Sheet

Basic Syntax

```
# This is a single-line comment

"""
This is a
multi-line comment
or docstring (used to describe a function or module)
"""

# Variables
variable_name = value

# Example
x = 5
y = "Hello"
```

Printing

```
print("Hello, World!")  # Print a string
print(x)  # Print the value of a variable
print(f"The value of x is {x}")  # f-string for formatted output
print("The value of y is", y)  # Print multiple values
```

Input from User

```
name = input("Enter your name: ")  # String input
age = int(input("Enter your age: "))  # Integer input
```

Data Types

- **int**: Integer numbers
- **float**: Floating-point numbers
- **str**: Strings (text)
- **bool**: Boolean values (True or False)
- **list**: Ordered, mutable collections
- **tuple**: Ordered, immutable collections
- **dict**: Key-value pairs
- **set**: Unordered, unique collections

```
# Examples
num = 10  # int
pi = 3.14  # float
name = "Alice"  # str
is_valid = True  # bool
```

```
colors = ["red", "green", "blue"]  # list
point = (1, 2)  # tuple
person = {"name": "Bob", "age": 30}  # dict
unique_numbers = {1, 2, 3}  # set
```

Memory Address

```
print(id(x))  # Get the memory address of a variable
```

Operators

- **Arithmetic Operators**: `+`, `-`, `*`, `/`, `//` (floor division), `%` (modulus), `**` (exponentiation)
- **Comparison Operators**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Logical Operators**: `and`, `or`, `not`
- **Assignment Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`

```
# Arithmetic
a = 10
b = 3
print(a + b)  # Addition
print(a - b)  # Subtraction
print(a * b)  # Multiplication
print(a / b)  # Division
print(a // b)  # Floor division
print(a % b)  # Modulus
print(a ** b)  # Exponentiation

# Comparison
print(a == b)  # Equals
print(a != b)  # Not equals
print(a > b)  # Greater than
print(a < b)  # Less than
print(a >= b)  # Greater than or equal to
print(a <= b)  # Less than or equal to

# Logical
print(a > 5 and b < 5)  # Logical AND
print(a > 5 or b > 5)  # Logical OR
print(not(a > 5))  # Logical NOT

# Assignment
a += 2  # a = a + 2
b *= 3  # b = b * 3
```

Looping

- **For Loop**

```
# Loop through a list
for color in colors:
    print(color)

# Loop through a range
for i in range(5):
    print(i)
```

- **While Loop**

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Conditional Statements

```
if condition:
    # code block
elif another_condition:
    # another code block
else:
    # another code block

# Example
age = 20
if age >= 18:
    print("Adult")
elif age >= 13:
    print("Teenager")
else:
    print("Child")
```

Example Program

```
# Get user input
name = input("Enter your name: ")
age = int(input("Enter your age: "))

# Check age and print message
```

```
if age >= 18:
    print(f"Hello, {name}. You are an adult.")
else:
    print(f"Hello, {name}. You are not an adult yet.")

# Loop through a range and print numbers
for i in range(1, 6):
    print(i)

# Define a list and print its elements
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

1. **`Integer (int)`**:

- Declaration: `num = 10`
- Printing: `print(num)`
- Getting from user: `num = int(input("Enter a number: "))`
- Casting: `num = int(value)`
- Printing data type and memory address:

```
print(type(num))
print(hex(id(num)))
```

2. **`Float (float)`**:

- Declaration: `float_num = 3.14`
- Printing: `print(float_num)`
- Getting from user: `float_num = float(input("Enter a float number: "))`
- Casting: `float_num = float(value)`
- Printing data type and memory address:

```
print(type(float_num))
print(hex(id(float_num)))
```

3. **`String (str)`**:

- Declaration: `text = "Hello, World!"`
- Printing: `print(text)`
- Getting from user: `text = input("Enter some text: ")`
- Casting: `text = str(value)`
- Printing data type and memory address:

```
print(type(text))
print(hex(id(text)))
```

4. **Boolean (bool)**:

- Declaration: `is_true = True`
- Printing: `print(is_true)`
- Getting from user: Boolean values are typically not directly taken from user input, but you can convert input to boolean if necessary.
- Casting: `is_true = bool(value)`
- Printing data type and memory address:

```
print(type(is_true))
print(hex(id(is_true)))
```

5. **List**:

- Declaration: `my_list = [1, 2, 3, 4, 5]`
- Printing: `print(my_list)`
- Getting from user: Not commonly taken directly from user input.
- Casting: `my_list = list(some_iterable)`
- Printing data type and memory address:

```
print(type(my_list))
print(hex(id(my_list)))
```

6. **Tuple**:

- Declaration: `my_tuple = (1, 2, 3, 4, 5)`
- Printing: `print(my_tuple)`
- Getting from user: Not commonly taken directly from user input.
- Casting: `my_tuple = tuple(some_iterable)`
- Printing data type and memory address:

```
print(type(my_tuple))
print(hex(id(my_tuple)))
```

7. **Dictionary (dict)**:

- Declaration: `my_dict = {'a': 1, 'b': 2, 'c': 3}`
- Printing: `print(my_dict)`
- Getting from user: Not commonly taken directly from user input.
- Casting: `my_dict = dict(some_mapping)`
- Printing data type and memory address:

```
print(type(my_dict))
print(hex(id(my_dict)))
```

1. Arithmetic Operators:

    - Addition: `+`
    - Subtraction: `-`
    - Multiplication: `*`
    - Division: `/`
    - Modulus (Remainder): `%`
    - Exponentiation: `**`

   Example:

```
a = 10
b = 3
print(a + b)  # Output: 13
print(a - b)  # Output: 7
print(a * b)  # Output: 30
print(a / b)  # Output: 3.333...
print(a % b)  # Output: 1
print(a ** b) # Output: 1000
```

2. Comparison Operators:

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

   Example:

```
x = 5
y = 7
print(x == y)  # Output: False
print(x != y)  # Output: True
print(x > y)   # Output: False
print(x < y)   # Output: True
print(x >= y)  # Output: False
print(x <= y)  # Output: True
```

3. Logical Operators:

    - AND: `and`
    - OR: `or`
    - NOT: `not`

   Example:

```
p = True
q = False
print(p and q)  # Output: False
print(p or q)   # Output: True
print(not p)    # Output: False
```

4. Assignment Operators:

- Assign value: `=`
- Add and assign: `+=`
- Subtract and assign: `-=`
- Multiply and assign: `*=`
- Divide and assign: `/=`
- Modulus and assign: `%=`
- Exponentiate and assign: `**=`

Example:

```
x = 10
x += 5  # Equivalent to x = x + 5
print(x)  # Output: 15
```

5. Membership Operators:

- `in`: Evaluates to True if a sequence contains a specified value.
- `not in`: Evaluates to True if a sequence does not contain a specified value.

Example:

```
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)     # Output: True
print(6 not in my_list) # Output: True
```

6. Identity Operators:

- `is`: Evaluates to True if both variables are the same object.
- `is not`: Evaluates to True if both variables are not the same object.

Example:

```
x = [1, 2, 3]
y = [1, 2, 3]
print(x is y)     # Output: False
print(x is not y) # Output: True
```

1. Looping in :

a. `for` Loop:

```
# Syntax:
for item in iterable:
    # Code block to execute for each item

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

b. `while` Loop:

```
# Syntax:
while condition:
    # Code block to execute as long as condition is True

# Example:
i = 0
while i < 5:
    print(i)
    i += 1
```

2. Advanced For Looping:

a. `enumerate()`:

```
# Syntax:
for index, item in enumerate(iterable):
    # Code block to execute with index and item

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

b. `zip()`:

```
# Syntax:
for item1, item2 in zip(iterable1, iterable2):
    # Code block to execute with corresponding items
```

```
# Example:
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
for name, age in zip(names, ages):
    print(name, age)
```

c. `range()`:

```
# Syntax:
for i in range(start, stop, step):
    # Code block to execute with each value of i

# Example:
for i in range(0, 10, 2):
    print(i)
```

d. Nested Loops:

```
# Syntax:
for item1 in iterable1:
    for item2 in iterable2:
        # Code block to execute with item1 and item2

# Example:
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for a in adj:
    for f in fruits:
        print(a, f)
```

e. `break` and `continue`:

```
# Example of break:
for i in range(10):
    if i == 5:
        break
    print(i)

# Example of continue:
for i in range(10):
    if i == 5:
        continue
    print(i)
```

Function Definition:

```
def function_name(parameters):
    """Docstring: Description of the function"""
    # Function body
    return result
```

Calling a Function:

```
result = function_name(arguments)
```

Parameters and Arguments:

```
def greet(name):
    print("Hello,", name)

greet("Alice")  # Argument: "Alice" is passed to the function greet
```

Default Parameters:

```
def greet(name="World"):
    print("Hello,", name)

greet()  # Output: Hello, World
greet("Alice")  # Output: Hello, Alice
```

Keyword Arguments:

```
def greet(name, message):
    print(message, name)

greet(message="Hi", name="Alice")  # Output: Hi Alice
```

Arbitrary Number of Arguments:

```
def calculate_sum(*args):
    return sum(args)

print(calculate_sum(1, 2, 3))  # Output: 6
```

Arbitrary Keyword Arguments:

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(key + ": " + value)

print_info(name="Alice", age="30")  # Output: name: Alice, age: 30
```

Return Statement:

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

result = add(3, 4)  # result = 7
```

Docstrings:

```
def square(num):
    """Returns the square of a number."""
    return num * num

print(square.__doc__)  # Output: Returns the square of a number.
```

Lambda Functions:

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

Decorators:

```
def decorator_function(func):
    def wrapper():
        print("Before function execution")
        func()
        print("After function execution")
    return wrapper

@decorator_function
def say_hello():
    print("Hello!")

say_hello()
```

```
# Output:
# Before function execution
# Hello!
# After function execution
```

Recursion:

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

print(factorial(5))  # Output: 120
```

1. **`len()`**: Returns the length of a string.

   ```
   my_string = "Hello, World!"
   print(len(my_string))  # Output: 13
   ```

2. **`lower()`**: Converts all characters in a string to lowercase.

   ```
   print(my_string.lower())  # Output: hello, world!
   ```

3. **`upper()`**: Converts all characters in a string to uppercase.

   ```
   print(my_string.upper())  # Output: HELLO, WORLD!
   ```

4. **`capitalize()`**: Converts the first character of a string to uppercase.

   ```
   print(my_string.capitalize())  # Output: Hello, world!
   ```

5. **`title()`**: Converts the first character of each word to uppercase.

   ```
   print(my_string.title())  # Output: Hello, World!
   ```

6. **`swapcase()`**: Swaps the case of each character in a string.

```
print(my_string.swapcase())  # Output: hELLO, wORLD!
```

7. **`strip()`**: Removes leading and trailing whitespace from a string.

```
my_string_with_spaces = "   Hello, World!   "
print(my_string_with_spaces.strip())  # Output: Hello, World!
```

8. **`replace()`**: Replaces occurrences of a substring within a string with another substring.

```
print(my_string.replace("Hello", "Hi"))  # Output: Hi, World!
```

9. **`count()`**: Returns the number of occurrences of a substring in a string.

```
print(my_string.count("o"))  # Output: 2
```

10. **`find()`**: Returns the lowest index of a substring in a string, or -1 if it's not found.

```
print(my_string.find("World"))  # Output: 7
```

11. **`startswith()`**: Checks if a string starts with a specified prefix.

```
print(my_string.startswith("Hello"))  # Output: True
```

12. **`endswith()`**: Checks if a string ends with a specified suffix.

```
print(my_string.endswith("World"))  # Output: False
```

13. **`split()`**: Splits a string into a list of substrings based on a delimiter.

```
print(my_string.split(","))  # Output: ['Hello', ' World!']
```

14. **`join()`**: Joins the elements of a list into a single string using a specified separator.

```
my_list = ["Hello", "World"]
print(", ".join(my_list))  # Output: Hello, World
```

15. **`isdigit()`**: Checks if all characters in a string are digits.

```
numeric_string = "123"
print(numeric_string.isdigit())  # Output: True
```

Access modifiers in control the accessibility of attributes and methods of a class. There are three main types of access modifiers in :

1. Public: Attributes and methods are accessible from outside the class.
2. Protected: Attributes and methods can be accessed within the class and its subclasses.
3. Private: Attributes and methods are accessible only within the class itself.

Let's explore these concepts with some real-time examples:

Public Access Modifier:

```
class Car:
    def __init__(self, make, model):
        self.make = make  # Public attribute
        self.model = model  # Public attribute

    def display_info(self):
        print(f"This is a {self.make} {self.model}.")

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

# Accessing public attributes and methods
print(my_car.make)  # Output: Toyota
print(my_car.model)  # Output: Camry
my_car.display_info()  # Output: This is a Toyota Camry.
```

In this example, `make` and `model` attributes are public, so they can be accessed from outside the class.

Protected Access Modifier:

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

    def display_info(self):
```

```
        print(f"Name: {self._name}, Age: {self._age}")

# Creating an instance of Person
person = Person("Alice", 30)

# Accessing protected attributes and methods
print(person._name)  # Output: Alice
print(person._age)  # Output: 30
person.display_info()  # Output: Name: Alice, Age: 30
```

Here, `_name` and `_age` attributes are marked as protected. Conventionally, they should be treated as non-public parts of the API, but they can still be accessed from outside the class.

Private Access Modifier:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

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

    def __update_balance(self, amount):  # Private method
        self.__balance += amount

# Creating an instance of BankAccount
account = BankAccount(1000)

# Accessing private attributes and methods (throws an error)
print(account.__balance)  # Error: 'BankAccount' object has no attribute '__balance'
account.display_balance()  # Output: Your balance: $1000
account.__update_balance(200)    #  Error:  'BankAccount'  object  has  no  attribute '__update_balance'
```

In this example, `__balance` attribute and `__update_balance()` method are marked as private. They cannot be accessed or modified directly from outside the class.

Classes and Objects:

In , a class is a blueprint for creating objects. It defines properties (attributes) and behaviors (methods) that objects of that class will have.

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def drive(self):
        print(f"{self.make} {self.model} is driving.")
```

```
# Creating objects of the Car class
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")

# Accessing object properties
print(car1.make, car1.model)  # Output: Toyota Corolla
print(car2.make, car2.model)  # Output: Honda Civic

# Calling object methods
car1.drive()  # Output: Toyota Corolla is driving.
car2.drive()  # Output: Honda Civic is driving.
```

Inheritance:

Inheritance allows a class (subclass) to inherit properties and behaviors from another class (superclass). It promotes code reusability.

```
class ElectricCar(Car):
    def __init__(self, make, model, battery_capacity):
        super().__init__(make, model)
        self.battery_capacity = battery_capacity

    def charge(self):
        print(f"{self.make} {self.model} is charging.")

# Creating objects of the ElectricCar class
electric_car = ElectricCar("Tesla", "Model S", "100 kWh")

# Accessing inherited properties and calling inherited methods
print(electric_car.make, electric_car.model)  # Output: Tesla Model S
electric_car.drive()  # Output: Tesla Model S is driving.

# Calling subclass method
electric_car.charge()  # Output: Tesla Model S is charging.
```

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It promotes flexibility and extensibility.

```
class Animal:
    def speak(self):
        pass

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

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

# Function that takes any Animal object and makes it speak
def make_speak(animal):
    print(animal.speak())

# Creating objects of Dog and Cat classes
dog = Dog()
cat = Cat()

# Making objects speak
make_speak(dog)  # Output: Woof!
make_speak(cat)  # Output: Meow!
```

Abstraction:

Abstraction is the process of hiding the implementation details and showing only the essential features of an object. In other words, it focuses on what an object does rather than how it does it.

Let's consider a simple example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Creating objects and calling the area method
rectangle = Rectangle(5, 4)
print("Area of rectangle:", rectangle.area())  # Output: 20
```

In this example, `Shape` is an abstract class with an abstract method `area()`. It defines a common interface for all shapes, but doesn't provide an implementation for the `area()` method. The `Rectangle` class inherits from `Shape` and provides its implementation of the `area()` method. Users of the `Rectangle` class don't need to know how the area is calculated; they just need to know that they can call the `area()` method to get the area of the rectangle.

Encapsulation:

Encapsulation is the bundling of data and methods that operate on the data into a single unit (class). It restricts direct access to some of the object's components and prevents external code from directly modifying an object's internal state.

```
class BankAccount:
    def __init__(self, balance=0):
        self._balance = balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"Deposited ${amount}. New balance: ${self._balance}")

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self._balance}")
        else:
            print("Insufficient funds")

# Creating an object of BankAccount
account = BankAccount(1000)

# Accessing and modifying balance directly (not encapsulated)
print(account._balance)  # Output: 1000
account._balance = 500   # Balance modified directly
print(account._balance)  # Output: 500

# Using methods to deposit and withdraw (encapsulated)
account.deposit(200)     # Output: Deposited $200. New balance: $700
account.withdraw(1000)   # Output: Withdrew $1000. New balance: $700
```

In this example, the `_balance` attribute is encapsulated within the `BankAccount` class. It can be accessed and modified only through the `deposit()` and `withdraw()` methods, ensuring that the internal state of the object remains consistent and preventing accidental misuse by external code.

Basic Concepts:

1. **Creating Lists:** You can create a list by enclosing comma-separated values within square brackets `[]`.

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

2. **Accessing Elements:** Elements in a list are accessed using index numbers, starting from 0.

```
print(my_list[0])  # Output: 1
```

3. **List Slicing:** You can extract a portion of a list using slicing.

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

4. **List Concatenation:** Lists can be concatenated using the `+` operator.

```
new_list = my_list + [6, 7, 8]
```

Advanced Concepts:

1. **List Comprehension:** A concise way to create lists.

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

2. **List Methods:**

   - `append()`: Adds an element to the end of the list.
   - `extend()`: Appends elements from another list.
   - `insert()`: Inserts an element at a specified position.
   - `remove()`: Removes the first occurrence of a value.
   - `pop()`: Removes and returns an element by index.
   - `index()`: Returns the index of the first occurrence of a value.
   - `count()`: Returns the number of occurrences of a value.
   - `sort()`: Sorts the list.
   - `reverse()`: Reverses the list.

```
my_list.append(6)
my_list.extend([7, 8])
my_list.insert(2, 2.5)
my_list.remove(3)
popped = my_list.pop(0)
index = my_list.index(5)
```

3. **List Membership:** Check if an element exists in a list.

```
print(5 in my_list)  # Output: True
```

4. **Nested Lists:** Lists can contain other lists.

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

Real-time Examples:

1. **Shopping List:**

```
shopping_list = ["Apples", "Bananas", "Milk", "Bread"]
```

2. **To-Do List:**

```
todo_list = ["Go to gym", "Buy groceries", "Finish report"]
```

3. **Matrix Operations:**

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

Basics of Tuples:

A tuple in is a collection of ordered elements, similar to a list, but it's immutable, meaning once it's created, you cannot change its contents. Tuples are defined using parentheses `()`.

Creating a Tuple:

```
my_tuple = (1, 2, 3, 4, 5)
```

Accessing Elements:

You can access elements of a tuple using indexing, similar to lists.

```
print(my_tuple[0])  # Output: 1
```

Iterating through a Tuple:

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

```
for item in my_tuple:
    print(item)
```

Tuple Packing and Unpacking:

Tuple packing is when you create a tuple without using parentheses.

```
my_tuple = 1, 2, 3
```

Tuple unpacking allows you to assign the elements of a tuple to multiple variables.

```
a, b, c = my_tuple
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Advanced Concepts and Functions:

Tuple Concatenation:

You can concatenate two tuples using the `+` operator.

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

Tuple Slicing:

You can slice tuples to extract specific subsets.

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

Tuple Methods:

Tuples have few methods because they are immutable. Two common methods are `count()` and `index()`.

- `count()`: Returns the number of times a specified value occurs in the tuple.
- `index()`: Returns the index of the first occurrence of a specified value.

```
my_tuple = (1, 2, 2, 3, 4, 5)
print(my_tuple.count(2))  # Output: 2
print(my_tuple.index(3))  # Output: 3
```

Real-life Example:

Consider storing the coordinates of points in a 2D space. Each point can be represented as a tuple `(x, y)`.

```
points = ((1, 2), (3, 4), (5, 6))

# Calculate the distance between two points
def distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5

print(distance(points[0], points[1]))  # Output: 2.8284271247461903
```

Basics of Sets in :

A set is an unordered collection of unique elements. It is defined using curly braces `{}` and elements are separated by commas. Sets do not allow duplicate elements.

Creating a Set:

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
print(my_set)  # Output: {1, 2, 3, 4, 5}

# Creating an empty set
empty_set = set()
```

Adding and Removing Elements:

```
# Adding elements to a set
my_set.add(6)
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

# Removing elements from a set
my_set.remove(3)
print(my_set)  # Output: {1, 2, 4, 5, 6}
```

Operations on Sets:

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# Union of two sets
union_set = set1.union(set2)
```

```
print(union_set)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}

# Intersection of two sets
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {4, 5}

# Difference of two sets
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2, 3}

# Subset check
is_subset = set1.issubset(set2)
print(is_subset)  # Output: False
```

Real-Life Example:

Consider a classroom with students who like different subjects. Each student has their own set of favorite subjects. We can represent this using sets in .

```
john = {"Math", "Science", "History"}
emma = {"Science", "Art", "English"}
sam = {"Math", "History", "Geography"}

# Intersection gives the subjects liked by all students
common_subjects = john.intersection(emma, sam)
print(common_subjects)  # Output: {'History', 'Science'}
```

Advanced Concepts:

Frozensets:

A frozenset is an immutable version of a set. Once created, its elements cannot be changed or modified.

```
frozen_set = frozenset([1, 2, 3, 4, 5])
```

Set Comprehensions:

Similar to list comprehensions, you can also create sets using set comprehensions.

```
squared_set = {x**2 for x in range(10)}
print(squared_set)  # Output: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}
```

More Functions:

 provides more functions to work with sets like `update()`, `discard()`, `clear()`, etc. You can explore them based on your requirements.

Real-Life Example:

Consider a social media platform where users can follow each other. You can represent followers of each user using sets.

```
john_followers = {"Emma", "Sam", "Alice"}
emma_followers = {"John", "Alice"}
sam_followers = {"John", "Alice", "Bob"}

# Union of all followers gives the total followers
total_followers = john_followers.union(emma_followers, sam_followers)
print(total_followers)  # Output: {'Sam', 'Bob', 'John', 'Emma', 'Alice'}
```

Basic Concepts:

1. Creating a Dictionary:

You can create a dictionary by enclosing comma-separated key-value pairs within curly braces `{}`.

```
# Creating a dictionary
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
```

2. Accessing Values:

You can access the value associated with a key using square brackets `[]`.

```
# Accessing values
print(my_dict['name'])  # Output: John
```

3. Adding and Updating Elements:

You can add new key-value pairs or update existing ones.

```
# Adding and updating elements
my_dict['email'] = 'john@example.com'
my_dict['age'] = 31
```

4. Removing Elements:

You can remove elements using `del` keyword or `pop()` method.

```
# Removing elements
del my_dict['city']
age = my_dict.pop('age')
```

5. Checking if Key Exists:

You can check if a key exists in a dictionary using the `in` keyword.

```
# Checking if key exists
if 'name' in my_dict:
    print("Key 'name' exists!")
```

Advanced Concepts:

1. Dictionary Methods:

- `keys()`: Returns a view of all keys in the dictionary.
- `values()`: Returns a view of all values in the dictionary.
- `items()`: Returns a view of all key-value pairs in the dictionary.

```
# Dictionary methods
keys = my_dict.keys()
values = my_dict.values()
items = my_dict.items()
```

2. Dictionary Comprehensions:

Like list comprehensions, you can also create dictionaries using dictionary comprehensions.

```
# Dictionary comprehension
squares = {x: x*x for x in range(1, 6)}
```

3. Nested Dictionaries:

Dictionaries can contain other dictionaries as values, allowing you to represent hierarchical data.

```
# Nested dictionaries
employee = {
    'name': 'John',
    'age': 30,
    'contact': {
        'email': 'john@example.com',
        'phone': '123-456-7890'
    }
}
```

4. Using Dictionaries for Counting:

Dictionaries are often used for counting occurrences of elements.

```
# Using dictionaries for counting
word_freq = {}
sentence = "hello world hello"
for word in sentence.split():
    word_freq[word] = word_freq.get(word, 0) + 1
```

5. Dictionary as Switch Case:

You can use dictionaries to implement switch-case like behavior.

```
# Dictionary as switch case
def func1():
    return "Function 1"

def func2():
    return "Function 2"

switch = {
    'case1': func1,
    'case2': func2,
}
result = switch['case1']()
```

Real-Time Examples:

1. **Building a Contact Book**: You can use dictionaries to build a contact book where each contact is represented by a dictionary with keys like name, email, phone, etc.
2. **Word Frequency Counter**: Analyze a text document and count the frequency of each word using a dictionary.
3. **Inventory Management**: Manage inventory items using dictionaries where each item has attributes like name, price, quantity, etc.
4. **Configuration Settings**: Store configuration settings of an application in a dictionary for easy access and manipulation.

Basic Concepts:

1. What is an Exception?

An exception is an error that occurs during the execution of a program. When an exception occurs, the program halts and raises an exception object.

2. Types of Exceptions:

- **Built-in Exceptions**: These are exceptions that are already defined in , such as `TypeError`, `ValueError`, `ZeroDivisionError`, etc.
- **User-defined Exceptions**: You can also define your own custom exceptions by creating a new class that inherits from the built-in `Exception` class.

3. The `try`, `except`, `else`, and `finally` Blocks:

- **`try`**: This block is used to wrap the code that may raise an exception.
- **`except`**: This block is used to handle the exception. You can specify which type of exception you want to catch, or catch all exceptions using a generic `except` block.
- **`else`**: This block is executed if the code in the `try` block doesn't raise any exceptions.
- **`finally`**: This block is always executed, regardless of whether an exception occurred or not. It's typically used for cleanup code (e.g., closing files or releasing resources).

Real-time Examples:

1. Division by Zero:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Error: Division by zero!")
```

2. File Handling:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    file.close()
except FileNotFoundError:
    print("Error: File not found!")
else:
    print("File content:", content)
finally:
    print("File handling completed.")
```

3. Custom Exception:

```
class CustomError(Exception):
    pass

def check_value(x):
    if x < 0:
        raise CustomError("Value must be positive")

try:
    check_value(-5)
except CustomError as e:
    print("Custom Error:", e)
```

4. Database Connection:

```
import sqlite3

try:
    connection = sqlite3.connect("example.db")
    cursor = connection.cursor()
    # Execute some SQL queries
except sqlite3.Error as e:
    print("Error connecting to database:", e)
finally:
    if connection:
        connection.close()
```

File I/O (Input/Output):

Reading from a File:

To read from a file in , you typically follow these steps:

1. **Open the File**: Use the `open()` function, specifying the file name and the mode (`'r'` for reading).
2. **Read the File Content**: Use methods like `read()`, `readline()`, or `readlines()` to read the content.
3. **Close the File**: Always close the file using the `close()` method to free up system resources.

Example:

```
# Open the file
file = open('example.txt', 'r')

# Read the content
content = file.read()
print(content)

# Close the file
file.close()
```

Writing to a File:

To write to a file in :

1. **Open the File**: Use the `open()` function with mode `'w'` for writing. This will create a new file if it doesn't exist or truncate the existing one.
2. **Write to the File**: Use the `write()` method to add content to the file.
3. **Close the File**: Close the file after writing to it.

Example:

```
# Open the file in write mode
file = open('example.txt', 'w')

# Write content to the file
file.write("Hello, world!\n")

# Close the file
file.close()
```

Date and Time Handling:

's `datetime` module provides classes for manipulating dates and times. Let's see some basic operations:

```
import datetime

# Get current date and time
now = datetime.datetime.now()
print("Current date and time:", now)

# Format date and time
formatted_now = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted date and time:", formatted_now)

# Parse a string to datetime
date_str = "2023-05-15"
parsed_date = datetime.datetime.strptime(date_str, "%Y-%m-%d")
print("Parsed date:", parsed_date)
```

Real-Time Examples:

File I/O Example:

Let's say we have a file `contacts.txt` containing names and emails. We can read this file, extract email addresses, and store them in a new file:

```
# Read from contacts.txt and extract emails
with open('contacts.txt', 'r') as f:
    emails = [line.split(',')[1].strip() for line in f]

# Write emails to a new file
with open('emails.txt', 'w') as f:
    for email in emails:
        f.write(email + '\n')
```

Date and Time Example:

Suppose you want to log events with timestamps. You can write a function to log messages with current timestamps to a file:

```
import datetime

def log_message(message):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    with open('log.txt', 'a') as f:  # 'a' for append mode
        f.write(f"{timestamp}: {message}\n")

# Example usage:
log_message("System initialized")
log_message("Error: File not found")
```

Basic Concepts:

1. **Creating an Array/List:**

   ```
   my_array = [1, 2, 3, 4, 5]
   ```

2. **Accessing Elements:** You can access elements using indices, starting from 0.

   ```
   print(my_array[0])  # Output: 1
   ```

3. **Array Length:** To find the length of an array, you can use the `len()` function.

   ```
   print(len(my_array))  # Output: 5
   ```

4. **Slicing:** Slicing allows you to access a subset of the array.

   ```
   print(my_array[1:3])  # Output: [2, 3]
   ```

Advanced Concepts:

1. **Appending and Extending:** You can add elements to an array using `append()` or add another array using `extend()`.

   ```
   my_array.append(6)
   print(my_array)  # Output: [1, 2, 3, 4, 5, 6]
   ```

```
another_array = [7, 8, 9]
my_array.extend(another_array)
print(my_array)  # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. **Removing Elements:** You can remove elements by value using `remove()` or by index using `del` or `pop()`.

```
my_array.remove(3)
print(my_array)  # Output: [1, 2, 4, 5, 6, 7, 8, 9]

del my_array[0]
print(my_array)  # Output: [2, 4, 5, 6, 7, 8, 9]

my_array.pop()  # Removes and returns the last element
print(my_array)  # Output: [2, 4, 5, 6, 7, 8]
```

3. **Searching:** You can search for elements using `index()` or `in` operator.

```
print(my_array.index(5))  # Output: 2
print(6 in my_array)  # Output: True
```

4. **Sorting and Reversing:** You can sort an array using `sort()` or reverse it using `reverse()`.

```
my_array.sort()
print(my_array)  # Output: [2, 4, 5, 6, 7, 8]

my_array.reverse()
print(my_array)  # Output: [8, 7, 6, 5, 4, 2]
```

Real-life Example: Student Grades

Let's say you want to store the grades of students in a class. You can use a array for this purpose.

```
grades = [85, 92, 78, 95, 88]

# Finding the average grade
average_grade = sum(grades) / len(grades)
print("Average Grade:", average_grade)

# Finding the highest and lowest grades
highest_grade = max(grades)
lowest_grade = min(grades)
print("Highest Grade:", highest_grade)
```

```
print("Lowest Grade:", lowest_grade)
```