Python Programming Cheat Sheet

Comments and Print Statement

In [ ]:
```
# This is a comment, comments help explain the code but are not executed
print("Hello, Data Analytics!")  # Prints the string "Hello, Data Analytics!" to
```

Variables and Data Types

In [ ]:
```
x = 5  # Assigning the integer 5 to the variable x
y = 2.5  # Assigning the float 2.5 to the variable y
name = "Alice"  # Assigning the string "Alice" to the variable name
is_analyst = True  # Assigning the boolean value True to the variable is_analyst

# The `type()` function is used to check the data type of a variable
print(type(x))  # Prints <class 'int'> because x is an integer
print(type(y))  # Prints <class 'float'> because y is a float
print(type(name))  # Prints <class 'str'> because name is a string
print(type(is_analyst))  # Prints <class 'bool'> because is_analyst is a boolean
```

Lists, Dictionaries, Tuples, and Sets

In [ ]:
```
data = [1, 2, 3, 4, 5]  # A list of integers
info = {'name': 'Alice', 'age': 25, 'role': 'Analyst'}  # A dictionary with keys
coordinates = (10.5, 20.7)  # A tuple representing coordinates, immutable (canno
unique_data = {1, 2, 3, 4, 5}  # A set of unique values, unordered and no duplic

# Print the data structures
print(data)  # Outputs the list: [1, 2, 3, 4, 5]
print(info)  # Outputs the dictionary: {'name': 'Alice', 'age': 25, 'role': 'Ana
print(coordinates)  # Outputs the tuple: (10.5, 20.7)
print(unique_data)  # Outputs the set: {1, 2, 3, 4, 5}
```

If-Else Conditional Statements

In [ ]:
```
x = 5  # Assigning the integer 5 to x
# Checking if x is greater than 0
if x > 0:
    print("Positive")  # If x is greater than 0, print "Positive"
else:
    print("Negative or Zero")  # If x is 0 or less, print "Negative or Zero"
```

For and While Loops

In [ ]:
```
# For loop: loops through a range of numbers from 0 to 4 (5 is not included)
for i in range(5):
    print(i)  # Prints the current value of i in each iteration

i = 0  # Initialize i to 0 for the while loop
# While loop: repeats as long as i is less than 5
while i < 5:
    print(i)  # Prints the current value of i
    i += 1  # Increases i by 1 in each iteration
```

Defining and Using Functions

In [ ]:
```
def greet(name):  # Define a function named greet, which takes a parameter 'name
    return f"Hello, {name}!"  # Return a greeting message with the passed name

# Call the function greet with the argument "Alice"
print(greet("Alice"))  # Outputs: "Hello, Alice!"
```

Understanding Functions in Python

Functions allow you to encapsulate logic for reusability, organization, and modularity.

In [3]:
```
# Defining a function using 'def' keyword, followed by the function name and par
def square(x):  # 'x' is the parameter that the function takes
    return x ** 2  # The function returns the square of 'x' (x multiplied by its

# Call the function with an argument, in this case, 5
result = square(5)  # The result will store the value returned by the square fun
print(result)  # Outputs: 25, since 5 squared is 25
```

```
25
```

Why Use Functions?

- **Modularity**: Break code into smaller chunks for better organization.
- **Reusability**: Write once, reuse multiple times.
- **Return Values**: Functions can return data, which you can use elsewhere in your program.

In [6]:
```
# Define a function that takes a name and returns a greeting message
def greet_person(name):  # 'name' is a parameter that accepts any string value
    return f"Hello, {name}!"  # The function returns a formatted string using f-

# Call the function and pass the string "Alice" as an argument
print(greet_person("Alice"))  # Outputs: "Hello, Alice!"
```

```
Hello, Alice!
```

Object-Oriented Programming (OOP) in Python

OOP helps you create blueprints (called classes) that define the structure and behavior of objects.

In [11]:
```python
# Defining a class named 'Person' that models a person's attributes and behavior
class Person:
    # __init__ is a constructor method that initializes the attributes of the ob
    def __init__(self, name, age):  # The class takes 'name' and 'age' as parame
        self.name = name  # 'self.name' refers to the name of the instance being
        self.age = age  # 'self.age' refers to the age of the instance being cre

    # A method inside the class that returns a greeting message
    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."

# Creating an object (instance) of the Person class
person1 = Person("Alice", 30)  # 'person1' is an object with name "Alice" and ag

# Call the 'greet' method on the object 'person1'
print(person1.greet())  # Outputs: "Hello, my name is Alice and I am 30 years ol
```

```
Hello, my name is Alice and I am 30 years old.
```

Understanding Objects in Python

Objects are like real-world things: they have **attributes** (properties) and can perform **actions** (methods). Think of objects as **blueprints** of something. In Python, these blueprints are defined by **classes**.

In [17]:
```python
# Example: Think of a 'Car' as a real-world object
# A Car object has properties like color, model, and speed, and it can perform a

# Let's create a class to define this blueprint for a Car object
class Car:
    def __init__(self, make, model, year):  # The constructor initializes attrib
        self.make = make  # 'self.make' is the attribute that stores the make of
        self.model = model  # 'self.model' stores the model of the car
        self.year = year  # 'self.year' stores the year of manufacture

    def start(self):  # Define a method (an action the car can perform)
        return f"The {self.year} {self.make} {self.model} has started!"  # Retur

    def stop(self):  # Another method to stop the car
        return f"The {self.year} {self.make} {self.model} has stopped."

# Creating an object (instance) of the Car class
my_car = Car("Toyota", "Corolla", 2020)  # 'my_car' is now a Car object with spe

# Call the 'start' and 'stop' methods on the my_car object
print(my_car.start())  # Outputs: "The 2020 Toyota Corolla has started!"
print(my_car.stop())   # Outputs: "The 2020 Toyota Corolla has stopped."
```

```
The 2020 Toyota Corolla has started!
The 2020 Toyota Corolla has stopped.
```

Intuition Behind Objects

- An **object** is like an **instance** of a blueprint (class).

- Objects are **specific** versions of things. In the above example, `my_car` is a specific car (a Toyota Corolla from 2020).
- Just like a real-world object, it has properties (make, model, year) and can perform actions (start, stop).

In [20]:
```
# Another example with a 'Dog' object to reinforce the concept

# Define a Dog class as a blueprint
class Dog:
    def __init__(self, name, breed):  # The Dog class has attributes: name and b
        self.name = name  # 'self.name' stores the name of the dog
        self.breed = breed  # 'self.breed' stores the breed of the dog

    def bark(self):  # Method to make the dog 'bark'
        return f"{self.name} is barking!"  # Returns a message indicating the do

    def sit(self):  # Method to make the dog 'sit'
        return f"{self.name} is sitting."

# Create an object (instance) of the Dog class
my_dog = Dog("Rex", "Labrador")  # 'my_dog' is a Labrador named Rex

# Call the methods on the my_dog object
print(my_dog.bark())  # Outputs: "Rex is barking!"
print(my_dog.sit())   # Outputs: "Rex is sitting."
```

```
Rex is barking!
Rex is sitting.
```

Key Takeaways About Objects:

- **Attributes**: Characteristics or properties that an object has. For a car, it could be make, model, year. For a dog, it could be name and breed.
- **Methods**: Actions that an object can perform. For a car, it could be starting or stopping. For a dog, it could be barking or sitting.
- Objects let you represent real-world things in a program and interact with them through their attributes and methods.

Why Use OOP?

- **Encapsulation**: Group data and methods together into objects.
- **Inheritance**: Create new classes from existing ones, reusing code efficiently.
- **Polymorphism**: Different classes can have methods with the same name but different implementations.

In [14]:
```
# Defining a new class 'Employee' that inherits from 'Person'
class Employee(Person):  # Employee class inherits all properties of Person
    def __init__(self, name, age, role):  # Adding an additional parameter 'role
        super().__init__(name, age)  # Inheriting 'name' and 'age' from the Pers
        self.role = role  # New attribute specific to Employee class
```

```
    # A method to introduce the employee
    def introduce(self):
        return f"I am {self.name}, a {self.role}."  # Use both inherited and new

# Creating an object (instance) of the Employee class
employee1 = Employee("Bob", 28, "Data Analyst")  # 'employee1' has name "Bob", a

# Call the 'greet' method (inherited from Person) and 'introduce' method
print(employee1.greet())  # Outputs: "Hello, my name is Bob and I am 28 years ol
print(employee1.introduce())  # Outputs: "I am Bob, a Data Analyst."
```

```
Hello, my name is Bob and I am 28 years old.
I am Bob, a Data Analyst.
```

Using NumPy and Pandas Libraries

In [ ]:

```
# Import NumPy for numerical operations and Pandas for data manipulation
import numpy as np  # np is an alias for NumPy
import pandas as pd  # pd is an alias for Pandas

# Creating a NumPy array with values 1, 2, and 3
arr = np.array([1, 2, 3])  # np.array creates an array from the list [1, 2, 3]

# Creating a Pandas DataFrame from a dictionary
df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})  # Creates a DataFrame with

# Print the NumPy array and Pandas DataFrame
print(arr)  # Outputs: [1 2 3]
print(df)  # Outputs the DataFrame with columns and rows
```

Data Selection, Filtering, and Aggregation

In [ ]:

```
# Select column 'col1' from the DataFrame df
selected_column = df['col1']  # Access column 'col1' of the DataFrame

# Filter the DataFrame for rows where 'col1' is greater than 1
filtered_df = df[df['col1'] > 1]  # Filters and returns rows where 'col1' > 1

# Calculate the mean of each column in the DataFrame
mean_values = df.mean()  # df.mean() returns the mean of numerical columns

# Print the results
print(selected_column)  # Outputs the values of 'col1': 1, 2
print(filtered_df)  # Outputs the filtered DataFrame where 'col1' > 1
print(mean_values)  # Outputs the mean of 'col1' and 'col2'
```

Reading and Writing CSV Files

In [ ]:

```
# Read a CSV file into a Pandas DataFrame
df = pd.read_csv('data.csv')  # pd.read_csv reads a CSV file named 'data.csv' in

# Write the DataFrame to a new CSV file, without including the index column
df.to_csv('output.csv', index=False)  # df.to_csv writes the DataFrame to 'outpu
```

```
# No output, but you can check the saved CSV file in the directory
```

Basic Plotting with Matplotlib

In [ ]:
```
# Import the Matplotlib library for plotting
import matplotlib.pyplot as plt  # plt is an alias for Matplotlib

# Plot a line graph with x-values [1, 2, 3] and y-values [4, 5, 6]
plt.plot([1, 2, 3], [4, 5, 6])  # Creates a line plot using the specified x and

# Add labels and title to the plot
plt.xlabel('X-axis')  # Label for the x-axis
plt.ylabel('Y-axis')  # Label for the y-axis
plt.title('Sample Plot')  # Title of the plot

# Display the plot
plt.show()  # Renders the plot in the output
```

In [ ]:
```

```

In [ ]:
```

```

Python Learning Resources

https://wiki.python.org/moin/BeginnersGuide/Programmers - Ultimate Python Resource by python.org

https://www.codecademy.com/courses/learn-python-3

https://www.youtube.com/watch?v=rfscVS0vtbw - Free Code Camp Video

https://app.datacamp.com/learn/courses/intro-to-python-for-data-science

END