Python Cheat Sheet

1. Basics

- **Comments:**

```
# Single-line comment
”””
Multi-line comment
”””
```

- **Variables:**

```
x = 10                    # Integer
y = 3.14                  # Float
name = ”Alice”            # String
is_active = True          # Boolean
```

---

2. Operators

- **Arithmetic**

```
a + b     # Addition
a - b     # Subtraction
a * b     # Multiplication
a / b     # Division
a % b     # Modulus
a ** b    # Exponentiation
```

- **Comparison**

```
a == b     # Equal to
a != b     # Not equal to
a > b      # Greater than
a < b      # Less than
```

- **Logical**

```
a and b    # Logical AND
a or b     # Logical OR
not a      # Logical NOT
```

3. Data Types

- **Numbers:**

```
int_var = 5
float_var = 5.5
```

- **Strings:**

```
s = ”Hello, World!”
length = len(s)
upper_s = s.upper()
lower_s = s.lower()
```

- **Lists:**

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

  - **Accessing Elements:**

```
first_element = my_list[0]          # 1
last_element = my_list[-1]          # 5
```

  - **Methods:**

```
my_list.append(6)                   # Add an element at the end
my_list.extend([7, 8])              # Add multiple elements
my_list.insert(0, 0)                    # Insert element at specified position
my_list.remove(3)                       # Remove first occurrence of value
popped_element = my_list.pop()          # Remove and return last element
my_list.clear()                         # Remove all elements
```

  - **List Comprehensions:**

```
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]
```

  - **Sorting and Reversing:**

```
my_list.sort()                          # Sort in place
my_list.sort(reverse=True)              # Sort in descending order
my_list.reverse()                       # Reverse the list
```

- **Tuples:**

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

  - **Accessing Elements:**

```
first_element = my_tuple[0]             # 1
```

  - **Methods:**

```
count = my_tuple.count(2)               # Count occurrences of value
index = my_tuple.index(3)               # Index of first occurrence of value
```

  - **Immutability:**

```
# Tuples cannot be modified
```

- **Dictionaries:**

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

  - **Accessing Elements:**

```
name = my_dict[”name”]                    # ”Alice”
```

  - **Methods:**

```
my_dict[”age”] = 26                       # Update value
my_dict[”city”] = ”NYC”                   # Add new key-value pair
value = my_dict.pop(”age”)                # Remove key and return value
del my_dict[”city”]                       # Remove key-value pair
keys = my_dict.keys()                     # Get dictionary keys
values = my_dict.values()                 # Get dictionary values
items = my_dict.items()                   # Get dictionary items (key-value pairs)
my_dict.clear()                           # Remove all items
```

  - **Dictionary Comprehensions:**

```
squares_dict = {x: x**2 for x in range(10)}
```

- **Sets:**

```
my_set = {1, 2, 3, 4, 5}
```

  - **Methods:**

```
my_set.add(6)                             # Add an element
my_set.remove(3)                          # Remove element (KeyError if not found)
my_set.discard(3)                         # Remove an element (no error if not found)
popped_element = my_set.pop()             # Remove and return an arbitrary element
my_set.clear()                            # Remove all elements
```

  - **Set Operations:**

```
a = {1, 2, 3}
b = {3, 4, 5}

union = a | b                             # {1, 2, 3, 4, 5}
intersection = a & b                      # {3}
difference = a - b                        # {1, 2}
```

```
symmetric_difference = a ^ b            # {1, 2, 4, 5}
```

4. Control Structures:

- ### If Statements:

```
if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is 10")
else:
    print("x is less than 10")
```

- ### For Loops:

```
for i in range(5):
    print(i)

# Iterating over a list
for fruit in fruits:
    print(fruit)
```

- ### While Loops:

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

- ### Break/Continue

```
for i in range(5):
    if i == 3:
        break                   # Exits loop
    if i == 1:
        continue                # Skips to the next iteration
        print(i)
```

5. Functions:

- **Defining Functions:**

```
def greet(name):
    return "Hello " + name


print(greet("Alice"))                  # Output: Hello Alice
```

- **Lambda Functions:**

```
add = lambda x, y: x + y
print(add(2, 3))                         # Output: 5
```

6. File Handling:

- **Reading a file:**

```
with open("file.txt", "r") as file:
    content = file.read()
```

- **Writing to a file:**

```
with open("file.txt", "w") as file:
    file.write("Hello, World!")
```

7. Exception Handling:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Execution completed.")
```

Classes and Objects in Python

1. Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software. Python is an OOP language, meaning you can create and manipulate objects in your programs.

- **Object:** An instance of a class. It has properties (attributes) and behaviors (methods).
- **Class:** A blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class will have.

2. Creating a Class

In Python, you define a class using the class keyword followed by the class name (by convention, class names start with an uppercase letter).

```
class Dog:
    pass  # A simple empty class
```

Here, Dog is a class with no attributes or methods.

3. Creating an Object (Instance of a Class)

You create an object by calling the class as if it were a function.

```
my_dog = Dog()            # Creating an object of class Dog
print(type(my_dog))       # Output: <class ’__main__.Dog’>
```

4. Attributes (Properties)

Attributes are variables that belong to a class. There are two types:

- **Instance Attributes:** Belong to the object itself.
- **Class Attributes:** Shared by all instances of the class.

Instance Attributes Example:

```
class Dog:
    def __init__(self, name, age):
        self.name = name      # Instance attribute
        self.age = age        # Instance attribute


my_dog = Dog(”Buddy”, 5)
```

```
print(my_dog.name)            # Output: Buddy
print(my_dog.age)             # Output: 5
```

- __init__(self, …): This is a special method called a **constructor**. It is automatically called when you create a new object of the class. The self parameter refers to the current instance of the class.

Class Attributes Example:

```
class Dog:
    species = ”Canis familiaris”          # Class attribute

    def __init__(self, name, age):
        self.name = name
        self.age = age

my_dog = Dog(”Buddy”, 5)
print(my_dog.species)                # Output: Canis familiaris
print(Dog.species)                   # Output: Canis familiaris
```

---

5. Methods (Behaviors)

Methods are functions defined within a class that describe the behaviors of an object.

```
class Dog:
    species = ”Canis familiaris”

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        return ”Woof!”

    def description(self):
        return f”{self.name} is {self.age} years old.”
```

```
my_dog = Dog(”Buddy”, 5)
print(my_dog.bark())        # Output: Woof!
print(my_dog.description())  # Output: Buddy is 5 years old.
```

- **self:** A reference to the current object instance. It’s how methods access the object’s attributes and other methods.

---

6. Inheritance

Inheritance allows a class to inherit attributes and methods from another class. The class being inherited from is called the **parent class**, and the class that inherits is called the **child class**.

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

    def speak(self):
        return ”Some sound”

class Dog(Animal):                 # Dog class inherits from Animal class
    def speak(self):
        return ”Woof!”

class Cat(Animal):                 # Cat class inherits from Animal class
    def speak(self):
        return ”Meow”

my_dog = Dog(”Buddy”)
my_cat = Cat(”Whiskers”)
print(my_dog.speak())              # Output: Woof!
print(my_cat.speak())              # Output: Meow
```

- **Overriding:** The ‘speak’ method in the ‘Dog’ and ‘Cat’ classes overrides the ‘speak’ method in the ‘Animal’ class.

---

7. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common parent class. It is often used in conjunction with inheritance.

```
for animal in [my_dog, my_cat]:
    print(animal.speak())
# Output:
# Woof!
# Meow
```

- Even though ‘my_dog’ and ‘my_cat’ are instances of different classes, they are both treated as ‘Animal’ objects when iterating.

---

8. Encapsulation

Encapsulation is the practice of hiding the internal state of an object and requiring all interaction to be performed through an object’s methods.

- **Public Attributes:** Accessible from outside the class.
- **Private Attributes:** Not accessible from outside the class. (Conventionally, these are prefixed with an underscore ‘_’).

```
class Dog:
    def __init__(self, name, age):
        self.name = name     # Public attribute
        self._age = age      # Private attribute

    def get_age(self):
        return self._age

my_dog = Dog(”Buddy”, 5)
print(my_dog.name)        # Output: Buddy
print(my_dog.get_age())   # Output: 5
```

- The ‘_age’ attribute is intended to be private, and accessing it directly is discouraged. Instead, use the ‘get_age()’ method.

---

9. Special Methods (Magic Methods)

Special methods (or magic methods) start and end with double underscores (‘__’). They allow you to define how objects of your class behave in certain operations (like addition, string conversion, etc.).

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

    def __str__(self):
        return f”Dog(name={self.name}, age={self.age})”

    def __add__(self, other):
        return self.age + other.age

dog1 = Dog(”Buddy”, 5)
dog2 = Dog(”Max”, 7)

print(dog1)                # Output: Dog(name=Buddy, age=5)
print(dog1 + dog2)         # Output: 12
```

- **__str__**: Defines how the object is printed.
- **__add__**: Defines behavior for the ‘+’ operator.

---

10. Class vs. Static Methods

- **Class Methods:** Use the “@classmethod decorator and take ‘cls’ as their first parameter. They can access and modify class state.
- **Static Methods:** Use the ‘@staticmethod’ decorator and don’t take ‘self’ or ‘cls’ as their first parameter. They behave like regular functions but belong to the class’s namespace.

```
class Dog:
    species = ”Canis familiaris”

    def __init__(self, name, age):
        self.name = name

        self.age = age


    @classmethod

    def set_species(cls, species):

        cls.species = species


    @staticmethod

    def bark_sound():

        return ”Woof!”


Dog.set_species(”Canis lupus familiaris”)

print(Dog.species)              # Output: Canis lupus familiaris

print(Dog.bark_sound())         # Output: Woof!
```

- **@classmethod**: Allows modification of class-level attributes.
- **@staticmethod**: Utility method that doesn’t access class or instance-specific data.

---

11. Conclusion

Classes and objects are foundational concepts in Python and OOP. Understanding them allows you to create modular, reusable, and organized code. Mastering these concepts will help you write more sophisticated programs and succeed in your exam.

---

prepared by chhavi-rohilla
https://www.linkedin.com/in/chhavi-rohilla-607996251