Python Object-Oriented Programming (OOP) - Complete Cheat Sheet

Table of Contents

Quick Reference

Basic Class Syntax

```python
class ClassName:
    # Class variable (shared by all instances)
    class_variable = "shared"

    def __init__(self, param1, param2):
        # Instance variables (unique to each instance)
        self.param1 = param1
        self.param2 = param2

    def instance_method(self):
        return f"{self.param1} {self.param2}"

    @classmethod
    def class_method(cls):
        return cls.class_variable

    @staticmethod
    def static_method():
        return "No access to class or instance"
```

Creating Objects

```python
obj = ClassName("value1", "value2")
obj.instance_method()
```

Four Pillars of OOP

1. ENCAPSULATION

Definition: Bundling data (attributes) and methods that operate on that data within a single unit (class), and restricting direct access to some components.

Real-World Example: Bank Account

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number  # Public
        self._branch = "Main Branch"          # Protected (convention)
        self.__balance = balance              # Private (name mangling)

    # Getter method
    def get_balance(self):
        return self.__balance

    # Setter method with validation
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"Deposited ${amount}. New balance: ${self.__balance}"
        return "Invalid amount"

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"Withdrawn ${amount}. Remaining balance: ${self.__balance}"
        return "Insufficient funds or invalid amount"

# Usage
account = BankAccount("ACC123", 1000)
print(account.get_balance())        # Output: 1000
print(account.deposit(500))         # Output: Deposited $500. New balance: $1500
# print(account.__balance)          # AttributeError: private variable
print(account._BankAccount__balance) # 1500 (name mangling - not recommended)
```

Key Points: - **Public** (`self.variable`): Accessible from anywhere - **Protected** (`self._variable`): Accessible within class and subclasses (convention only) - **Private** (`self.__variable`): Accessible only within the class (name mangling)

Properties (Pythonic Way)

```
class Employee:
    def __init__(self, name, salary):
```

```
        self._name = name
        self._salary = salary

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, value):
        if value > 0:
            self._salary = value
        else:
            raise ValueError("Salary must be positive")

    @salary.deleter
    def salary(self):
        print("Deleting salary")
        del self._salary

# Usage
emp = Employee("John", 50000)
print(emp.salary)      # Calls getter
emp.salary = 60000     # Calls setter
```

---

2. INHERITANCE

Definition: A mechanism where a new class (child/derived) inherits properties and behaviors from an existing class (parent/base).

Real-World Example: Vehicle System

```
# Base/Parent Class
class Vehicle:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.is_running = False

    def start(self):
        self.is_running = True
        return f"{self.brand} {self.model} is starting..."

    def stop(self):
        self.is_running = False
```

```
        return f"{self.brand} {self.model} has stopped."

    def info(self):
        return f"{self.year} {self.brand} {self.model}"

# Derived/Child Class
class Car(Vehicle):
    def __init__(self, brand, model, year, num_doors):
        super().__init__(brand, model, year)  # Call parent constructor
        self.num_doors = num_doors

    def open_trunk(self):
        return f"The trunk of {self.model} is opened."

    # Method overriding
    def info(self):
        parent_info = super().info()
        return f"{parent_info} - {self.num_doors} doors"

class Motorcycle(Vehicle):
    def __init__(self, brand, model, year, has_sidecar):
        super().__init__(brand, model, year)
        self.has_sidecar = has_sidecar

    def wheelie(self):
        return f"{self.model} is doing a wheelie!"

# Usage
car = Car("Toyota", "Camry", 2023, 4)
bike = Motorcycle("Harley-Davidson", "Street 750", 2023, False)

print(car.start())        # Inherited method
print(car.open_trunk())   # Car-specific method
print(car.info())         # Overridden method

print(bike.wheelie())     # Motorcycle-specific method
```

Types of Inheritance

```
# 1. Single Inheritance
class Parent:
    pass

class Child(Parent):
    pass
```

```python
# 2. Multiple Inheritance
class Father:
    def skills(self):
        return "Programming"

class Mother:
    def skills(self):
        return "Cooking"

class Child(Father, Mother):  # MRO: Child -> Father -> Mother
    pass

child = Child()
print(child.skills())  # Output: Programming (Father's method)

# 3. Multilevel Inheritance
class GrandParent:
    def family_name(self):
        return "Smith"

class Parent(GrandParent):
    def home(self):
        return "New York"

class Child(Parent):
    def hobby(self):
        return "Gaming"

# 4. Hierarchical Inheritance
class Animal:
    pass

class Dog(Animal):
    pass

class Cat(Animal):
    pass

# 5. Hybrid Inheritance (combination of multiple types)
class A:
    pass

class B(A):
    pass

class C(A):
```

```
    pass

class D(B, C):
    pass
```

Method Resolution Order (MRO)

```
class A:
    def method(self):
        return "A"

class B(A):
    def method(self):
        return "B"

class C(A):
    def method(self):
        return "C"

class D(B, C):
    pass

d = D()
print(d.method())        # Output: B
print(D.mro())           # Shows the order: [D, B, C, A, object]
print(D.__mro__)         # Same as above (tuple format)
```

---

3. POLYMORPHISM

Definition: The ability of different objects to respond to the same method call in different ways.

Types of Polymorphism

1. Method Overriding (Runtime Polymorphism)

```
class Animal:
    def speak(self):
        return "Some sound"

    def move(self):
        return "Moves somehow"

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

```
        return "Woof! Woof!"

    def move(self):
        return "Runs on four legs"

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

    def move(self):
        return "Walks gracefully"

class Bird(Animal):
    def speak(self):
        return "Chirp! Chirp!"

    def move(self):
        return "Flies in the sky"

# Polymorphism in action
def animal_behavior(animal):
    print(f"Sound: {animal.speak()}")
    print(f"Movement: {animal.move()}")
    print("-" * 30)

# Usage
animals = [Dog(), Cat(), Bird()]

for animal in animals:
    animal_behavior(animal)

# Output:
# Sound: Woof! Woof!
# Movement: Runs on four legs
# ------------------------------
# Sound: Meow!
# Movement: Walks gracefully
# ------------------------------
# Sound: Chirp! Chirp!
# Movement: Flies in the sky
```

2. Duck Typing (Python-specific)

```
# "If it walks like a duck and quacks like a duck, it's a duck"

class Duck:
```

```
    def swim(self):
        return "Duck is swimming"

    def fly(self):
        return "Duck is flying"

class Airplane:
    def swim(self):
        return "Airplane floats on water"

    def fly(self):
        return "Airplane is flying"

class Whale:
    def swim(self):
        return "Whale is swimming"

def in_the_water(entity):
    print(entity.swim())

# Works with any object that has swim() method
in_the_water(Duck())      # Duck is swimming
in_the_water(Airplane())  # Airplane floats on water
in_the_water(Whale())     # Whale is swimming
```

3. Operator Overloading

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

```
    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)

# Usage
v1 = Vector(2, 3)
v2 = Vector(4, 1)

print(v1 + v2)    # Vector(6, 4)
print(v1 - v2)    # Vector(-2, 2)
print(v1 * 3)     # Vector(6, 9)
print(v1 == v2)   # False
```

Real-World Example: Payment System

```
class Payment:
    def process_payment(self, amount):
        raise NotImplementedError("Subclass must implement this method")

class CreditCardPayment(Payment):
    def __init__(self, card_number):
        self.card_number = card_number

    def process_payment(self, amount):
        return f"Processing ${amount} via Credit Card ending in {self.card_number[-4:]}"

class PayPalPayment(Payment):
    def __init__(self, email):
        self.email = email

    def process_payment(self, amount):
        return f"Processing ${amount} via PayPal account: {self.email}"

class CryptoPayment(Payment):
    def __init__(self, wallet_address):
        self.wallet_address = wallet_address

    def process_payment(self, amount):
        return f"Processing ${amount} via Crypto wallet: {self.wallet_address[:10]}..."

# Unified payment processing
def checkout(payment_method, amount):
    print(payment_method.process_payment(amount))

# Usage
payments = [
```

```
    CreditCardPayment("1234-5678-9012-3456"),
    PayPalPayment("user@example.com"),
    CryptoPayment("0x742d35Cc6634C0532925a3b844Bc9e7595f0bEb")
]

for payment in payments:
    checkout(payment, 100)
```

---

4. ABSTRACTION

Definition: Hiding complex implementation details and showing only the necessary features of an object.

Abstract Base Classes (ABC)

```
from abc import ABC, abstractmethod

# Abstract class
class Shape(ABC):
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def area(self):
        """Calculate area - must be implemented by subclasses"""
        pass

    @abstractmethod
    def perimeter(self):
        """Calculate perimeter - must be implemented by subclasses"""
        pass

    # Concrete method (non-abstract)
    def description(self):
        return f"This is a {self.color} shape"

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

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

```
    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, color, radius):
        super().__init__(color)
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

# Usage
# shape = Shape("red")  # TypeError: Can't instantiate abstract class

rect = Rectangle("blue", 5, 3)
circle = Circle("red", 4)

print(rect.area())          # 15
print(rect.perimeter())     # 16
print(rect.description())   # This is a blue shape

print(circle.area())        # 50.26544
print(circle.perimeter())   # 25.13272
```

Real-World Example: Database Operations

```
from abc import ABC, abstractmethod

class Database(ABC):
    @abstractmethod
    def connect(self):
        pass

    @abstractmethod
    def disconnect(self):
        pass

    @abstractmethod
    def execute_query(self, query):
        pass

class MySQLDatabase(Database):
    def connect(self):
```

```
        return "Connected to MySQL database"

    def disconnect(self):
        return "Disconnected from MySQL database"

    def execute_query(self, query):
        return f"Executing MySQL query: {query}"

class MongoDBDatabase(Database):
    def connect(self):
        return "Connected to MongoDB database"

    def disconnect(self):
        return "Disconnected from MongoDB database"

    def execute_query(self, query):
        return f"Executing MongoDB query: {query}"

class PostgreSQLDatabase(Database):
    def connect(self):
        return "Connected to PostgreSQL database"

    def disconnect(self):
        return "Disconnected from PostgreSQL database"

    def execute_query(self, query):
        return f"Executing PostgreSQL query: {query}"

# Usage
def perform_database_operations(db: Database):
    print(db.connect())
    print(db.execute_query("SELECT * FROM users"))
    print(db.disconnect())
    print("-" * 50)

# Works with any database implementation
databases = [MySQLDatabase(), MongoDBDatabase(), PostgreSQLDatabase()]

for db in databases:
    perform_database_operations(db)
```

Advanced Concepts

Magic Methods (Dunder Methods)

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        # Informal string representation (for print)
        return f"'{self.title}' by {self.author}"

    def __repr__(self):
        # Official string representation (for debugging)
        return f"Book('{self.title}', '{self.author}', {self.pages})"

    def __len__(self):
        return self.pages

    def __eq__(self, other):
        return self.title == other.title and self.author == other.author

    def __lt__(self, other):
        return self.pages < other.pages

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

    def __contains__(self, keyword):
        return keyword.lower() in self.title.lower()

# Usage
book1 = Book("Python Basics", "John Doe", 300)
book2 = Book("Advanced Python", "Jane Smith", 450)

print(book1)              # 'Python Basics' by John Doe
print(repr(book1))        # Book('Python Basics', 'John Doe', 300)
print(len(book1))         # 300
print(book1 < book2)      # True
print(book1 + book2)      # 750
print("Python" in book1)  # True
```

Common Magic Methods

Method	Description	Example
`__init__`	Constructor	`obj = MyClass()`
`__str__`	Informal string	`str(obj)` or `print(obj)`
`__repr__`	Official string	`repr(obj)`
`__len__`	Length	`len(obj)`
`__getitem__`	Index access	`obj[key]`
`__setitem__`	Set item	`obj[key] = value`
`__delitem__`	Delete item	`del obj[key]`
`__contains__`	Membership	`item in obj`
`__iter__`	Iterator	`for item in obj`
`__next__`	Next item	`next(obj)`
`__add__`	Addition	`obj1 + obj2`
`__sub__`	Subtraction	`obj1 - obj2`
`__mul__`	Multiplication	`obj * num`
`__eq__`	Equality	`obj1 == obj2`
`__lt__`	Less than	`obj1 < obj2`
`__gt__`	Greater than	`obj1 > obj2`
`__call__`	Call object	`obj()`
`__enter__`	Context manager entry	`with obj:`
`__exit__`	Context manager exit	`with obj:`

Class Methods vs Static Methods vs Instance Methods

```python
class MyClass:
    class_variable = "I'm a class variable"

    def __init__(self, value):
        self.instance_variable = value

    # Instance method (most common)
    def instance_method(self):
        return f"Instance method called. Value: {self.instance_variable}"

    # Class method
    @classmethod
    def class_method(cls):
        return f"Class method called. Class variable: {cls.class_variable}"

    # Alternative constructor
    @classmethod
    def from_string(cls, string_value):
        value = int(string_value)
        return cls(value)

    # Static method (doesn't access instance or class)
```

```
    @staticmethod
    def static_method():
        return "Static method called. No access to class or instance."

    @staticmethod
    def utility_function(x, y):
        return x + y

# Usage
obj = MyClass(10)

print(obj.instance_method())              # Requires instance
print(MyClass.class_method())             # Can call on class
print(MyClass.static_method())            # Can call on class
print(MyClass.utility_function(5, 3))     # 8

# Alternative constructor
obj2 = MyClass.from_string("42")
print(obj2.instance_variable)             # 42
```

Composition (Has-A Relationship)

```
# Engine class
class Engine:
    def __init__(self, horsepower, type):
        self.horsepower = horsepower
        self.type = type

    def start(self):
        return f"{self.type} engine with {self.horsepower}HP started"

# Wheel class
class Wheel:
    def __init__(self, size):
        self.size = size

# Car class using composition
class Car:
    def __init__(self, brand, engine, wheel_size):
        self.brand = brand
        self.engine = engine  # Car HAS-A Engine
        self.wheels = [Wheel(wheel_size) for _ in range(4)]  # Car HAS-A Wheels

    def start(self):
        return f"{self.brand} car: {self.engine.start()}"
```

```
    def wheel_info(self):
        return f"Car has 4 wheels of size {self.wheels[0].size} inches"

# Usage
engine = Engine(300, "V6")
car = Car("Toyota", engine, 18)

print(car.start())        # Toyota car: V6 engine with 300HP started
print(car.wheel_info())   # Car has 4 wheels of size 18 inches
```

Property Decorators

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero is not possible")
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self._celsius = (value - 32) * 5/9

    @property
    def kelvin(self):
        return self._celsius + 273.15

# Usage
temp = Temperature(25)
print(temp.celsius)      # 25
print(temp.fahrenheit)   # 77.0
print(temp.kelvin)       # 298.15

temp.fahrenheit = 98.6
print(temp.celsius)      # 37.0
```

Dataclasses (Python 3.7+)

```python
from dataclasses import dataclass, field
from typing import List

@dataclass
class Product:
    name: str
    price: float
    quantity: int = 0
    tags: List[str] = field(default_factory=list)

    def total_value(self):
        return self.price * self.quantity

    def __post_init__(self):
        if self.price < 0:
            raise ValueError("Price cannot be negative")

# Usage
product = Product("Laptop", 999.99, 5, ["electronics", "computers"])
print(product)                    # Product(name='Laptop', price=999.99, ...)
print(product.total_value())      # 4999.95
```

Interview Questions & Answers

Beginner Level

Q1: What is the difference between a class and an object?

A: A class is a blueprint or template for creating objects, while an object is an instance of a class.

```python
# Class (blueprint)
class Dog:
    def __init__(self, name):
        self.name = name

# Objects (instances)
dog1 = Dog("Buddy")
dog2 = Dog("Max")
```

Q2: Explain the `self` keyword in Python.

A: `self` represents the instance of the class and is used to access instance variables and methods. It's automatically passed as the first parameter to instance

methods.

```
class Person:
    def __init__(self, name):
        self.name = name  # self refers to the current instance

    def greet(self):
        return f"Hello, I'm {self.name}"  # Accessing instance variable
```

Q3: What is the difference between `__init__` and `__new__`?

A: - `__new__` is called first to create the instance (rarely overridden) - `__init__` is called after to initialize the instance

```
class Example:
    def __new__(cls):
        print("Creating instance")
        return super().__new__(cls)

    def __init__(self):
        print("Initializing instance")

obj = Example()
# Output:
# Creating instance
# Initializing instance
```

Q4: What are class variables vs instance variables?

A: - **Class variables**: Shared among all instances - **Instance variables**: Unique to each instance

```
class Student:
    school_name = "ABC School"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

s1 = Student("John")
s2 = Student("Jane")

print(s1.school_name)  # ABC School (shared)
print(s2.school_name)  # ABC School (shared)
print(s1.name)         # John (unique)
print(s2.name)         # Jane (unique)
```

Q5: Explain method overriding with an example.

A: Method overriding occurs when a child class provides its own implementation of a method that exists in the parent class.

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):  # Overriding
        return "Bark"

class Cat(Animal):
    def sound(self):  # Overriding
        return "Meow"

dog = Dog()
cat = Cat()
print(dog.sound())  # Bark
print(cat.sound())  # Meow
```

Intermediate Level

Q6: What is the difference between `@staticmethod` and `@classmethod`?

A: - `@classmethod`: Receives the class as first argument (`cls`), can access/modify class state - `@staticmethod`: Doesn't receive class or instance, just a utility function

```
class Math:
    pi = 3.14159

    @classmethod
    def circle_area(cls, radius):
        return cls.pi * radius ** 2  # Can access class variable

    @staticmethod
    def add(x, y):
        return x + y  # No access to class or instance

print(Math.circle_area(5))  # 78.53975
print(Math.add(3, 4))       # 7
```

Q7: Explain Multiple Inheritance and MRO (Method Resolution Order).

A: Multiple inheritance allows a class to inherit from multiple parent classes. MRO determines the order in which base classes are searched when looking for a method.

```
class A:
    def method(self):
        return "A"

class B(A):
    def method(self):
        return "B"

class C(A):
    def method(self):
        return "C"

class D(B, C):
    pass

d = D()
print(d.method())  # B (follows C3 linearization)
print(D.mro())     # [D, B, C, A, object]
```

Q8: What is the purpose of `super()`?

A: `super()` returns a proxy object that allows you to call methods of the parent class, enabling proper method resolution in inheritance hierarchies.

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

    def greet(self):
        return f"Hello from {self.name}"

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Call parent's __init__
        self.age = age

    def greet(self):
        parent_greet = super().greet()  # Call parent's greet
        return f"{parent_greet}, I'm {self.age} years old"
```

```python
child = Child("John", 10)
print(child.greet())
# Hello from John, I'm 10 years old
```

Q9: Explain the concept of duck typing in Python.

A: Duck typing means that the type or class of an object is less important than the methods it defines. "If it walks like a duck and quacks like a duck, it's a duck."

```python
class Duck:
    def swim(self):
        return "Duck swimming"

class Fish:
    def swim(self):
        return "Fish swimming"

class Car:
    def drive(self):
        return "Car driving"

def make_it_swim(entity):
    return entity.swim()  # Doesn't check type, just calls swim()

print(make_it_swim(Duck()))  # Works
print(make_it_swim(Fish()))  # Works
# print(make_it_swim(Car()))  # AttributeError: no swim()
```

Q10: What are abstract classes and when would you use them?

A: Abstract classes cannot be instantiated and are used to define a common interface for subclasses. They ensure that derived classes implement specific methods.

```python
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

    @abstractmethod
    def refund(self, transaction_id):
        pass
```

```python
class StripeProcessor(PaymentProcessor):
    def process_payment(self, amount):
        return f"Processing ${amount} via Stripe"

    def refund(self, transaction_id):
        return f"Refunding transaction {transaction_id} via Stripe"

# payment = PaymentProcessor()  # TypeError: Can't instantiate
stripe = StripeProcessor()      # Works
```

Advanced Level

Q11: Explain the difference between `__str__` and `__repr__`.

A: - `__str__`: Returns a user-friendly string representation (for end users) - `__repr__`: Returns an unambiguous string representation (for developers/debugging)

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Point at ({self.x}, {self.y})"

    def __repr__(self):
        return f"Point({self.x}, {self.y})"

p = Point(3, 4)
print(str(p))   # Point at (3, 4)
print(repr(p))  # Point(3, 4)
print(p)        # Uses __str__ if available, else __repr__
```

Q12: What is a metaclass? Provide an example.

A: A metaclass is a class of a class that defines how a class behaves. Classes are instances of metaclasses.

```python
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
```

```
        return cls._instances[cls]

class Database(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = "Connected"

db1 = Database()
db2 = Database()
print(db1 is db2)  # True (same instance - Singleton pattern)
```

Q13: Explain the concept of composition vs inheritance.

A: - **Inheritance (is-a)**: Subclass is a type of parent class - **Composition (has-a)**: Class contains instances of other classes

“‘python # Inheritance (is-a) class Animal: pass

class Dog(Animal): # Dog IS-AN Animal pass

Composition (has-a)