Python Object-Oriented Programming Cheat Sheet

Prof. Laio Oriel Seman - laio.seman@ufsc.br

🎓 Understanding Classes and Objects

What is a Class?

A class is like a blueprint or template for creating objects

💡 *Think of a class as a cookie cutter - it defines the shape and structure*

Class: Car

Attributes (Data):
- brand
- model
- year
- color

Methods (Behavior):
- start()
- stop()
- honk()

Creates Objects:

car1	car2	car3
brand: Toyota model: Camry year: 2020 color: Blue	brand: Honda model: Civic year: 2019 color: Red	brand: Ford model: Focus year: 2021 color: White

1 Define the Class

Create the blueprint with attributes and methods

```
class Car:
    def __init__(self, brand, model, year, color):
        self.brand = brand
        self.model = model
        self.year = year
        self.color = color

    def start(self):
        return f'{self.brand} {self.model} is starting!'

    def honk(self):
        return 'Beep beep!'
```

2 Create Objects (Instantiation)

Use the class to create actual objects

```
# Creating objects from the Car class
my_car = Car('Toyota', 'Camry', 2020, 'Blue')
friend_car = Car('Honda', 'Civic', 2019, 'Red')

# Each object has its own data
print(my_car.brand)     # Toyota
print(friend_car.brand) # Honda
```

3 Use Object Methods

Call methods on your objects

```
# Using methods on objects
print(my_car.start())    # Toyota Camry is starting!
print(friend_car.honk()) # Beep beep!

# Objects are independent
my_car.color = 'Green'   # Only changes my_car
print(my_car.color)      # Green
print(friend_car.color)  # Red (unchanged)
```

🏗 Core OOP Concepts

Classes & Objects

Blueprint for creating objects with attributes and methods

- Class is a template/blueprint
- Object is an instance of a class
- Attributes store data
- Methods define behavior

Inheritance

Create new classes based on existing classes

- Child class inherits from parent
- super() calls parent methods
- Method overriding
- Multiple inheritance supported

Encapsulation

Hide internal implementation details using private/protected members

- Private: __attribute (name mangling)
- Protected: _attribute (convention)
- Public: attribute (default)
- Property decorators for getters/setters

Polymorphism

Same interface, different implementations

- Method overriding
- Duck typing
- Abstract base classes
- Same method name, different behavior

🔮 Special Methods (Magic Methods)

Constructor & Destructor

Methods:

`__init__(self, ...)`	Constructor - initializes new instance
`__new__(cls, ...)`	Creates new instance (before __init__)
`__del__(self)`	Destructor - called when object is garbage collected

Example:

```
class Resource:
    def __init__(self, name):
        self.name = name
        print(f'Resource {name} created')

    def __del__(self):
        print(f'Resource {self.name} destroyed')

# Usage
r = Resource('Database')
del r  # Explicitly delete
```

String Representation

Methods:

`__str__(self)`	Human-readable string (print, str())
`__repr__(self)`	Developer-friendly representation
`__format__(self, spec)`	Custom formatting

Example:

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

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

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

    def __format__(self, spec):
        if spec == 'coords':
            return f'x={self.x}, y={self.y}'
        return str(self)

p = Point(3, 4)
print(str(p))     # (3, 4)
```

Arithmetic Operators

Methods:

`__add__(self, other)`	Addition (+)
`__sub__(self, other)`	Subtraction (-)
`__mul__(self, other)`	Multiplication (*)
`__truediv__(self, other)`	Division (/)
`__eq__(self, other)`	Equality (==)
`__lt__(self, other)`	Less than (<)

Example:

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

    def __str__(self):
```

Container Methods

Methods:

`__len__(self)`	Length (len())
`__getitem__(self, key)`	Get item (obj[key])
`__setitem__(self, key, value)`	Set item (obj[key] = value)
`__contains__(self, item)`	Membership (in operator)
`__iter__(self)`	Make object iterable

Example:

```
class CustomList:
    def __init__(self):
        self._items = []

    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value

    def __contains__(self, item):
        return item in self._items

    def __iter__(self):
        return iter(self._items)
```

🎨 Design Patterns

Singleton Pattern

Ensure only one instance of a class exists

Use Cases: Database connections, Logger, Configuration

```
class Singleton:
    _instance = None
    _initialized = False

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not self._initialized:
            self.value = 0
            self._initialized = True

    def increment(self):
        self.value += 1

# Usage
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True (same instance)
s1.increment()
print(s2.value)  # 1
```

Factory Pattern

Create objects without specifying exact classes

Use Cases: Object creation logic, Plugin systems, Dynamic loading

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

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

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

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == 'dog':
            return Dog()
        elif animal_type == 'cat':
            return Cat()
        else:
            raise ValueError(f'Unknown animal: {animal_type}')

# Usage
factory = AnimalFactory()
dog = factory.create_animal('dog')
```

Observer Pattern

Notify multiple objects about state changes

Use Cases: Event systems, Model-View architectures, Notifications

```
class Subject:
    def __init__(self):
        self._observers = []
        self._state = None

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self)

    def set_state(self, state):
        self._state = state
        self.notify()

    @property
    def state(self):
        return self._state

class Observer:
    def __init__(self, name):
        self.name = name

    def update(self, subject):
```

Decorator Pattern

Add functionality to objects dynamically

Use Cases: Adding features, Middleware, Function enhancement

```
from abc import ABC, abstractmethod

class Coffee(ABC):
    @abstractmethod
    def cost(self):
        pass

    @abstractmethod
    def description(self):
        pass

class SimpleCoffee(Coffee):
    def cost(self):
        return 2.0

    def description(self):
        return 'Simple coffee'

class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

class MilkDecorator(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 0.5

    def description(self):
        return self._coffee.description() + ', milk'
```

🚀 Advanced Concepts

Metaclasses

Classes that create classes

```
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 DatabaseConnection(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = 'Connected'

# Usage
db1 = DatabaseConnection()
db2 = DatabaseConnection()
print(db1 is db2)  # True
```

Descriptors

Control attribute access with __get__, __set__, __delete__

```
class ValidatedAttribute:
    def __init__(self, validator):
        self.validator = validator
        self.name = None

    def __set_name__(self, owner, name):
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        if self.validator(value):
            obj.__dict__[self.name] = value
        else:
            raise ValueError(f'Invalid value for {self.name}')

class Person:
    age = ValidatedAttribute(lambda x: isinstance(x, int) and x >= 0)

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

# Usage
p = Person('Alice', 30)
# p.age = -5  # Would raise ValueError
```

Dataclasses

Simplified class creation for data storage

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

@dataclass
class Person:
    name: str
    age: int
    email: str = ''
    hobbies: List[str] = field(default_factory=list)

    def __post_init__(self):
        if self.age < 0:
            raise ValueError('Age cannot be negative')

@dataclass(frozen=True)  # Immutable
class Point:
    x: float
    y: float

# Usage
person = Person('Alice', 30)
print(person)  # Person(name='Alice', age=30, email='', hobbies=[])

point = Point(1.0, 2.0)
# point.x = 3.0  # Would raise FrozenInstanceError
```

Context Managers

Manage resources with __enter__ and __exit__

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        print(f'Opening file {self.filename}')
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f'Closing file {self.filename}')
        if self.file:
            self.file.close()
        if exc_type:
            print(f'Exception occurred: {exc_val}')
        return False  # Don't suppress exceptions

# Usage
with FileManager('test.txt', 'w') as f:
    f.write('Hello, World!')
# File automatically closed
```

📊 OOP Principles Summary

Principle	Definition	Key Benefit	Python Implementation
Encapsulation	Hide internal implementation details	Data protection & modularity	`Private attributes (__attr), properties`
Inheritance	Create new classes from existing ones	Code reuse & hierarchy	`class Child(Parent), super()`
Polymorphism	Same interface, different implementations	Flexibility & extensibility	`Method overriding, duck typing`
Abstraction	Hide complexity, show only essentials	Simplified interfaces	`Abstract base classes (ABC)`

💡 Best Practices & Tips

Design Principles

- 🔶 Use descriptive class and method names
- 🔶 Follow the Single Responsibility Principle
- 🔶 Prefer composition over inheritance when possible
- 🔶 Use abstract base classes for interfaces
- 🔶 Implement __str__ and __repr__ for better debugging

Implementation Tips

- 🔶 Use properties instead of getters/setters
- 🔶 Follow PEP 8 naming conventions
- 🔶 Document your classes with docstrings
- 🔶 Use type hints for better code clarity
- 🔶 Keep methods small and focused

🔧 Quick Reference

```
# Class definition template
class MyClass:
    def __init__(self, param):
        self.param = param

    def __str__(self):
        return f'MyClass({self.param})'

    def method(self):
        return self.param

# Inheritance
class Child(Parent):
    def __init__(self, param, extra):
        super().__init__(param)
        self.extra = extra
```

🏗 Master OOP to build scalable and maintainable applications!

Encapsulation • Inheritance • Polymorphism • Abstraction