Object-Oriented Programming (OOP) in Python Cheat Sheet

1. Classes and Objects

Class Definition:

```
class MyClass:
    def __init__(self, value):
        self.value = value

    def display_value(self):
        print(self.value)
```

Creating an Object:

```
obj = MyClass(10)
obj.display_value()  # Output: 10
```

2. Attributes and Methods

Instance Attributes:

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

    def bark(self):
        print(f"{self.name} says woof!")
```

Class Attributes:

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

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

Properties in Python

Properties in Python provide a way to manage the access to instance attributes. They allow you to define methods that get and set the value of an attribute, while still using attribute access syntax.

Using Properties for Attribute Access

Basic Property Example:

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

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not value:
            raise ValueError("Name cannot be empty")
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value < 0:
            raise ValueError("Age cannot be negative")
        self._age = value

person = Person("Alice", 30)
print(person.name)  # Output: Alice
person.name = "Bob"
print(person.name)  # Output: Bob
```

Read-Only Property:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @property
    def area(self):
        return 3.14 * self._radius ** 2

circle = Circle(5)
print(circle.radius)  # Output: 5
print(circle.area)    # Output: 78.5
```

Computed Property:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        if value <= 0:
            raise ValueError("Width must be positive")
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if value <= 0:
            raise ValueError("Height must be positive")
        self._height = value

    @property
    def area(self):
        return self._width * self._height

rect = Rectangle(4, 5)
print(rect.area)  # Output: 20
rect.width = 6
print(rect.area)  # Output: 30
```

Deleting a Property:

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

    @property
    def name(self):
        return self._name

    @name.deleter
    def name(self):
        del self._name

person = Person("Alice")
print(person.name)  # Output: Alice
del person.name
# print(person.name)  # This will raise an AttributeError
```

3. Inheritance

Single Inheritance:

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

    def speak(self):
        raise NotImplementedError("Subclasses must implement this
method")

class Dog(Animal):
    def speak(self):
        return f"{self.name} says woof!"
```

Multiple Inheritance:

```
class Canine:
    def bark(self):
        return "Woof!"

class Pet:
    def play(self):
        return "Playing!"

class Dog(Canine, Pet):
    pass

dog = Dog()
print(dog.bark())  # Output: Woof!
print(dog.play())  # Output: Playing!
```

4. Encapsulation

Private Attributes:

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

    def get_make(self):
        return self.__make

    def get_model(self):
        return self.__model
```

Protected Attributes:

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

```
        self._make = make
        self._model = model
```

5. Polymorphism

Method Overriding:

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this
method")

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

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

animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())
```

Method Overloading (Not natively supported, but can be mimicked):

```
class Math:
    def add(self, a, b, c=None):
        if c:
            return a + b + c
        else:
            return a + b

math = Math()
print(math.add(1, 2))       # Output: 3
print(math.add(1, 2, 3))    # Output: 6
```

6. Abstraction

Abstract Base Classes:

```
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
```

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

rect = Rectangle(3, 4)
print(rect.area())  # Output: 12
```

7. Composition

Using Objects as Attributes:

```python
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()

car = Car()
print(car.start())  # Output: Engine started
```

8. SOLID Principles

Single Responsibility Principle:

The Single Responsibility Principle states that a class should have only one reason to change, meaning it should have only one job or responsibility.

Explanation:

A class should only have one responsibility or one reason to change. This principle helps in making the system easier to understand, maintain, and refactor.

In this example, UserInfo handles user data, UserAuth handles authentication, and UserProfile handles displaying user profile information. Each class has a single responsibility.

```python
class UserInfo:
    def __init__(self, username, password):
        self.username = username
        self.password = password

class UserAuth:
    def __init__(self, user_info):
        self.user_info = user_info

    def authenticate(self, password):
        return self.user_info.password == password

class UserProfile:
    def __init__(self, user_info):
```

```
        self.user_info = user_info

    def display_profile(self):
        return f"Username: {self.user_info.username}"
```

Open/Closed Principle:

The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Explanation:

You should be able to add new functionality to a class without changing its existing code. This principle helps in making the system more flexible and easier to extend.

In this example, the Shape class is open for extension (you can add new shapes like Circle and Rectangle), but closed for modification (you don't need to change the Shape class to add new shapes).

```
class Shape:
    def area(self):
        raise NotImplementedError("Subclasses must implement this method")

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

    def area(self):
        return 3.14 * self.radius ** 2

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

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

def calculate_area(shapes):
    for shape in shapes:
        print(shape.area())

shapes = [Circle(5), Rectangle(4, 6)]
calculate_area(shapes)
```

Liskov Substitution Principle:

The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Explanation:

Subclasses should be substitutable for their base classes. This principle ensures that a subclass can stand in for its superclass without causing errors or unexpected behavior.

In this example, Sparrow can replace Bird without any issues, but Penguin violates the Liskov Substitution Principle because it cannot fly, which is expected behavior for a Bird.

```
class Bird:
    def fly(self):
        raise NotImplementedError("Subclasses must implement this method")

class Sparrow(Bird):
    def fly(self):
        return "Sparrow is flying"

class Penguin(Bird):
    def fly(self):
        raise Exception("Penguins cannot fly")

def make_bird_fly(bird):
    return bird.fly()

sparrow = Sparrow()
print(make_bird_fly(sparrow))  # Output: Sparrow is flying

penguin = Penguin()
# print(make_bird_fly(penguin))  # This will raise an Exception
```

Interface Segregation Principle:

The Interface Segregation Principle states that no client should be forced to depend on methods it does not use.

Explanation:

Clients should not be forced to implement interfaces they do not use. This principle helps in creating more focused and cohesive interfaces.

In this example, BasicPrinter only implements the Printer interface, while AllInOnePrinter implements both Printer and Scanner interfaces. This way, BasicPrinter is not forced to implement the scan_document method, adhering to the Interface Segregation Principle.

```
from abc import ABC, abstractmethod

class Printer(ABC):
    @abstractmethod
    def print_document(self, document):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan_document(self, document):
```

```
        pass

class BasicPrinter(Printer):
    def print_document(self, document):
        return f"Printing: {document}"

class AllInOnePrinter(Printer, Scanner):
    def print_document(self, document):
        return f"Printing: {document}"

    def scan_document(self, document):
        return f"Scanning: {document}"
```

Dependency Inversion Principle:

The Dependency Inversion Principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend on details. Details should depend on abstractions.

Explanation:

High-level modules should not depend on low-level modules; both should depend on abstractions. This principle helps in reducing the coupling between different modules of the system.

In this example, the Application class depends on the DatabaseConnection abstraction rather than a specific database implementation. This allows the Application class to work with any database that implements the DatabaseConnection interface, adhering to the Dependency Inversion Principle.

```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def disconnect(self):
        pass

class MySQLDatabase(DatabaseConnection):
    def connect(self):
        return "MySQL Database connected"

    def disconnect(self):
        return "MySQL Database disconnected"

class PostgreSQLDatabase(DatabaseConnection):
    def connect(self):
        return "PostgreSQL Database connected"

    def disconnect(self):
        return "PostgreSQL Database disconnected"
```

```python
class Application:
    def __init__(self, database: DatabaseConnection):
        self.database = database

    def start(self):
        return self.database.connect()

    def stop(self):
        return self.database.disconnect()

mysql_db = MySQLDatabase()
app = Application(mysql_db)
print(app.start())  # Output: MySQL Database connected

postgres_db = PostgreSQLDatabase()
app = Application(postgres_db)
print(app.start())  # Output: PostgreSQL Database connected
```

9. Design Patterns

Singleton Pattern:

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls,
*args, **kwargs)
        return cls._instance

singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2)  # Output: True
```

Factory Pattern:

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

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

class AnimalFactory:
    @staticmethod
    def get_animal(animal_type):
        if animal_type == "dog":
            return Dog()
        elif animal_type == "cat":
            return Cat()
        else:
```

```python
            return None

dog = AnimalFactory.get_animal("dog")
print(dog.speak())  # Output: Woof!
```

Observer Pattern:

```python
class Subject:
    def __init__(self):
        self._observers = []

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

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

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

class Observer:
    def update(self, message):
        raise NotImplementedError("Subclasses must implement this 
method")

class ConcreteObserver(Observer):
    def update(self, message):
        print(f"Received message: {message}")

subject = Subject()
observer = ConcreteObserver()
subject.attach(observer)
subject.notify("Hello, Observer!")
```

Decorator Pattern:

```python
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper

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

say_hello()
# Output:
# Before function call
# Hello!
# After function call
```

Strategy Pattern:

```
class Strategy:
    def execute(self, a, b):
        raise NotImplementedError("Subclasses must implement this
method")

class Addition(Strategy):
    def execute(self, a, b):
        return a + b

class Subtraction(Strategy):
    def execute(self, a, b):
        return a - b

class Context:
    def __init__(self, strategy):
        self._strategy = strategy

    def set_strategy(self, strategy):
        self._strategy = strategy

    def execute_strategy(self, a, b):
        return self._strategy.execute(a, b)

context = Context(Addition())
print(context.execute_strategy(5, 3))  # Output: 8
context.set_strategy(Subtraction())
print(context.execute_strategy(5, 3))  # Output: 2
```

Adapter Pattern:

```
class EuropeanSocket:
    def voltage(self):
        return 230

    def live(self):
        return 1

    def neutral(self):
        return -1

class AmericanSocket:
    def voltage(self):
        return 120

    def live(self):
        return 1

    def neutral(self):
        return 0

class Adapter(AmericanSocket):
    def __init__(self, european_socket):
        self.european_socket = european_socket
```

```python
    def voltage(self):
        return self.european_socket.voltage()

    def live(self):
        return self.european_socket.live()

    def neutral(self):
        return self.european_socket.neutral()

european_socket = EuropeanSocket()
adapter = Adapter(european_socket)
print(adapter.voltage())  # Output: 230
```

Facade Pattern:

```python
class CPU:
    def freeze(self):
        print("CPU freezing")

    def jump(self, position):
        print(f"CPU jumping to {position}")

    def execute(self):
        print("CPU executing")

class Memory:
    def load(self, position, data):
        print(f"Memory loading {data} at {position}")

class HardDrive:
    def read(self, lba, size):
        return f"Reading {size} bytes from {lba}"

class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hard_drive = HardDrive()

    def start(self):
        self.cpu.freeze()
        self.memory.load(0, self.hard_drive.read(0, 1024))
        self.cpu.jump(0)
        self.cpu.execute()

computer = ComputerFacade()
computer.start()
```

Command Pattern:

```python
class Command:
    def execute(self):
        raise NotImplementedError("Subclasses must implement this
```

```
method")

class Light:
    def on(self):
        print("Light is on")

    def off(self):
        print("Light is off")

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.off()

class RemoteControl:
    def __init__(self):
        self.commands = {}

    def set_command(self, button, command):
        self.commands[button] = command

    def press_button(self, button):
        if button in self.commands:
            self.commands[button].execute()

light = Light()
remote = RemoteControl()
remote.set_command("on", LightOnCommand(light))
remote.set_command("off", LightOffCommand(light))
remote.press_button("on")  # Output: Light is on
remote.press_button("off")  # Output: Light is off
```

Proxy Pattern:

```
class RealSubject:
    def request(self):
        return "RealSubject: Handling request."

class Proxy:
    def __init__(self, real_subject):
        self._real_subject = real_subject

    def request(self):
        if self.check_access():
            return self._real_subject.request()
```

```
        else:
            return "Proxy: Access denied."

    def check_access(self):
        # Simulate access control
        return True

real_subject = RealSubject()
proxy = Proxy(real_subject)
print(proxy.request())  # Output: RealSubject: Handling request.
```

Flyweight Pattern:

```
class Flyweight:
    _instances = {}

    def __new__(cls, shared_state):
        if shared_state not in cls._instances:
            cls._instances[shared_state] = super(Flyweight,
cls).__new__(cls)
            cls._instances[shared_state].shared_state = shared_state
        return cls._instances[shared_state]

fw1 = Flyweight("shared")
fw2 = Flyweight("shared")
fw3 = Flyweight("unique")
print(fw1 is fw2)  # Output: True
print(fw1 is fw3)  # Output: False
```

Chain of Responsibility Pattern:

```
class Handler:
    def __init__(self, successor=None):
        self._successor = successor

    def handle(self, request):
        if self._successor:
            self._successor.handle(request)

class HandlerA(Handler):
    def handle(self, request):
        if request == "A":
            print("HandlerA handled request")
        else:
            super().handle(request)

class HandlerB(Handler):
    def handle(self, request):
        if request == "B":
            print("HandlerB handled request")
        else:
            super().handle(request)

handler_chain = HandlerA(HandlerB())
```

```
handler_chain.handle("A")  # Output: HandlerA handled request
handler_chain.handle("B")  # Output: HandlerB handled request
handler_chain.handle("C")  # No output
```

Mediator Pattern:

```
class Mediator:
    def notify(self, sender, event):
        raise NotImplementedError("Subclasses must implement this
method")

class ConcreteMediator(Mediator):
    def __init__(self):
        self._colleague_a = None
        self._colleague_b = None

    def set_colleague_a(self, colleague):
        self._colleague_a = colleague

    def set_colleague_b(self, colleague):
        self._colleague_b = colleague

    def notify(self, sender, event):
        if event == "A":
            print("Mediator reacts on A and triggers B")
            self._colleague_b.do_b()
        elif event == "B":
            print("Mediator reacts on B and triggers A")
            self._colleague_a.do_a()

class Colleague:
    def __init__(self, mediator):
        self._mediator = mediator

class ColleagueA(Colleague):
    def do_a(self):
        print("ColleagueA does A")
        self._mediator.notify(self, "A")

class ColleagueB(Colleague):
    def do_b(self):
        print("ColleagueB does B")
        self._mediator.notify(self, "B")

mediator = ConcreteMediator()
colleague_a = ColleagueA(mediator)
colleague_b = ColleagueB(mediator)
mediator.set_colleague_a(colleague_a)
mediator.set_colleague_b(colleague_b)

colleague_a.do_a()
# Output:
# ColleagueA does A
```

```
# Mediator reacts on A and triggers B
# ColleagueB does B
```

Memento Pattern:

```
class Memento:
    def __init__(self, state):
        self._state = state

    def get_state(self):
        return self._state

class Originator:
    def __init__(self, state):
        self._state = state

    def save_state(self):
        return Memento(self._state)

    def restore_state(self, memento):
        self._state = memento.get_state()

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

    def get_state(self):
        return self._state

originator = Originator("State1")
memento = originator.save_state()
originator.set_state("State2")
print(originator.get_state())  # Output: State2
originator.restore_state(memento)
print(originator.get_state())  # Output: State1
```

Visitor Pattern:

```
class Visitor:
    def visit_element_a(self, element):
        raise NotImplementedError("Subclasses must implement this
method")

    def visit_element_b(self, element):
        raise NotImplementedError("Subclasses must implement this
method")

class Element:
    def accept(self, visitor):
        raise NotImplementedError("Subclasses must implement this
method")

class ElementA(Element):
    def accept(self, visitor):
        visitor.visit_element_a(self)
```

```python
class ElementB(Element):
    def accept(self, visitor):
        visitor.visit_element_b(self)

class ConcreteVisitor(Visitor):
    def visit_element_a(self, element):
        print("Visiting ElementA")

    def visit_element_b(self, element):
        print("Visiting ElementB")

elements = [ElementA(), ElementB()]
visitor = ConcreteVisitor()
for element in elements:
    element.accept(visitor)
# Output:
# Visiting ElementA
# Visiting ElementB
```

Template Method Pattern:

```python
class TemplateMethod:
    def execute(self):
        self.step1()
        self.step2()
        self.step3()

    def step1(self):
        raise NotImplementedError("Subclasses must implement this
method")

    def step2(self):
        raise NotImplementedError("Subclasses must implement this
method")

    def step3(self):
        raise NotImplementedError("Subclasses must implement this
method")

class ConcreteClassA(TemplateMethod):
    def step1(self):
        print("ConcreteClassA: Step 1")

    def step2(self):
        print("ConcreteClassA: Step 2")

    def step3(self):
        print("ConcreteClassA: Step 3")

class ConcreteClassB(TemplateMethod):
    def step1(self):
        print("ConcreteClassB: Step 1")
```

```
    def step2(self):
        print("ConcreteClassB: Step 2")

    def step3(self):
        print("ConcreteClassB: Step 3")

# Use the Template Method pattern
objA = ConcreteClassA()
objA.execute()
# Output:
# ConcreteClassA: Step 1
# ConcreteClassA: Step 2
# ConcreteClassA: Step 3

objB = ConcreteClassB()
objB.execute()
# Output:
# ConcreteClassB: Step 1
# ConcreteClassB: Step 2
# ConcreteClassB: Step 3
```

10. Mixins

Using Mixins to Add Functionality:

```
class LogMixin:
    def log(self, message):
        print(f"Log: {message}")

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

class Dog(Animal, LogMixin):
    def bark(self):
        self.log(f"{self.name} says woof!")

dog = Dog("Buddy")
dog.bark()  # Output: Log: Buddy says woof!
```

11. Metaclasses

Creating a Custom Metaclass:

```
class Meta(type):
    def __new__(cls, name, bases, dct):
        dct['class_name'] = name
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=Meta):
    pass
```

```python
obj = MyClass()
print(obj.class_name)  # Output: MyClass
```

12. Descriptors

Using Descriptors for Attribute Management:

```python
class Descriptor:
    def __init__(self, name=None):
        self.name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]

class MyClass:
    attr = Descriptor('attr')

    def __init__(self, attr):
        self.attr = attr

obj = MyClass(10)
print(obj.attr)  # Output: 10
obj.attr = 20
print(obj.attr)  # Output: 20
```

13. Context Managers

Creating a Custom Context Manager:

```python
class MyContextManager:
    def __enter__(self):
        print("Entering context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")

with MyContextManager():
    print("Inside context")
# Output:
# Entering context
# Inside context
# Exiting context
```

14. Decorators

Class Decorators:

```
def decorator(cls):
    class Wrapped(cls):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            print(f"Instance of {cls.__name__} created")
    return Wrapped

@decorator
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(10)
# Output: Instance of MyClass created
```