Module 3 & 4

Created by	Karan Maurya
Status	Not started

Topics

Python Cheatsheet: Module 3 + Module 4

(Advanced + Intermediate Friendly, Highly Structured)

Module 3: Functions, Modules, and Packages

3.1 Functions

Core Concepts:

- **Functions** encapsulate reusable blocks of code.
- **Python functions are first-class objects** (they can be passed as arguments, returned, assigned to variables).

Defining Functions

```
def greet(name):
    return f"Hello, {name}"
```

- `return` ends function execution and outputs a value.
- Functions **without a return** implicitly return `None`.

Arguments Types

Type	Syntax	Description
Positional	`func(a, b)`	Matched by order
Keyword	`func(a=10, b=20)`	Matched by name
Default	`def func(a=0)`	Defaults if missing
Variable-length	`*args, **kwargs`	Many args or keyword args

```
def example(*args, **kwargs):
    print(args)
    print(kwargs)

example(1,2,3, name="Alice", age=25)
```

Output:

```
(1, 2, 3)
{'name': 'Alice', 'age': 25}
```

> Advanced Insight: Prefer keyword-only arguments in complex APIs:

```
def connect(*, host, port):
```

3.2 Lambda (Anonymous) Functions

- Single-expression mini functions.

```
square = lambda x: x*x
print(square(5))  # 25
```

Useful for **short-term, inline** operations, often in:

- `sorted()`
- `map()`
- `filter()`
- `reduce()`

Example:

```
nums = [1,2,3,4]
print(list(map(lambda x: x*2, nums)))
# Output: [2, 4, 6, 8]
```

3.3 `map()`, `filter()`, `reduce()`

Function	Purpose	Example
`map(func, iterable)`	Apply function to all items	`map(lambda x: x*2, nums)`
`filter(func, iterable)`	Select items where func(item) is True	`filter(lambda x: x>2, nums)`
`reduce(func, iterable)`	Apply func cumulatively	`reduce(lambda x,y: x+y, nums)`

```
from functools import reduce
print(reduce(lambda x, y: x+y, [1,2,3,4]))  # 10
```

3.4 Modules and Importing

- **Module** = `.py` file containing functions, classes, variables.
- **Package** = directory with multiple modules and `__init__.py`.

Importing styles:

```
import math
from math import sqrt
from math import pi as PI
```

> Pro Tip:
>
> Import only necessary parts to reduce namespace pollution.

Custom Module Example:

```
# utils.py
def add(x, y):
    return x + y

# main.py
from utils import add
print(add(2, 3))
```

Module 4: Object-Oriented Programming (OOP) in Python

4.1 Classes and Objects

Defining a Class

```
class Car:
    def __init__(self, brand):
```

```
        self.brand = brand

    def drive(self):
        print(f"{self.brand} is driving!")

# Creating Object
c = Car("Toyota")
c.drive()
```

Output:

```
Toyota is driving!
```

- `__init__` is the constructor method.
- `self` points to the current object instance.

4.2 Attributes and Methods

Type	Description	Example
Instance Attribute	Unique to each object	`self.color`
Class Attribute	Shared among all objects	`Car.wheels = 4`

```
class Car:
    wheels = 4  # Class attribute

    def __init__(self, brand):
        self.brand = brand  # Instance attribute
```

> Advanced Note: Use class attributes for constants like MAX_SPEED, shared limits etc.

4.3 Inheritance

- Inherit attributes and methods from a parent class.

```
class ElectricCar(Car):
    def __init__(self, brand, battery):
        super().__init__(brand)
        self.battery = battery
```

- `super()` is used to call parent class methods/constructor.

> Best Practice:
>
> Always prefer `super()` rather than explicitly naming the parent class. This ensures maintainability during refactoring.

4.4 Method Overriding

- Redefining a parent method in the child class.

```
class ElectricCar(Car):
    def drive(self):
        print(f"{self.brand} drives silently.")
```

4.5 Encapsulation (Private Attributes)

- Convention for private attributes: prefix with `_` (protected) or `__` (name mangling).

```
class BankAccount:
    def __init__(self):
        self.__balance = 0  # Private

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

> Deep Insight:
>
> Python doesn't truly enforce private, it's just a naming convention to avoid accidental access.

4.6 Polymorphism

- Same method name behaves differently across classes.

Example:

```
class Bird:
    def sound(self):
        print("Chirp")

class Dog:
    def sound(self):
        print("Bark")

def make_sound(animal):
    animal.sound()

make_sound(Bird())  # Chirp
make_sound(Dog())   # Bark
```

> Real-World Usage:
>
> Polymorphism is heavily used in design patterns like Strategy, Factory, and Decorator.

Summary for Module 3 + 4

Topic	Key Focus	Special Insights
Functions	First-class objects	Prefer pure functions for safety
Lambda	Lightweight operations	Use in functional constructs

Modules	Code organization	Import cleanly, not greedily
Classes	Blueprint for objects	Encapsulation best practices
Inheritance	Code reuse	Always use `super()` smartly
Polymorphism	Method overriding	Enables design pattern implementation

This concludes Module 3 + 4 cheatsheet!

(Optimized for fast recall, technical understanding, and real-world application.)