upGrad

Lecture Notes:
Advanced Python Programming

Advanced Functional Programming

Types of Arguments

Think of argument types as **tools in a contract**:

- What must be passed?
- What can vary?
- How strict should you be?

Type	Syntax	Purpose
Positional	`f(10, 20)`	Arguments assigned based on order
Keyword	`f(x=10, y=20)`	Explicit name-value mapping
Default	`def f(x=0)`	Provide fallback if not passed
Variable Positional	`*args`	Capture extra positional args as tuple
Variable Keyword	`**kwargs`	Capture extra keyword args as dict
Positional-only	`def f(x, /)`	Forces argument to be passed by position
Keyword-only	`def f(*, y)`	Forces argument to be passed by keyword
Functions as arguments	`func1(func2, <args>)`	Allows passing a function as an argument to another function

Ask an LLM

- How do `*args` and `**kwargs` support extensible design?
- When would you enforce keyword-only or positional-only arguments?
- How does Python resolve function calls with multiple argument types?
- Can default arguments be mutable? Why is that dangerous?

Functional Tools in Python

`map()`: Transform Elements

Applies a function to each item in an iterable and returns an iterator.

Syntax:

```python
map(function, iterable)
```

Example:

```python
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x ** 2, nums))  # [1, 4, 9, 16]
```

Use when transforming data column-wise or vectorising logic.

`filter()`: Select Elements Conditionally

Keeps only those elements where the function returns `True`.

Example:

```python
nums = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, nums))  # [2, 4]
```

Use to filter rows, remove nulls, or subset by condition.

`reduce()`: Collapse into One Value

Cumulative application of a binary function to an iterable.

Syntax:

```python
from functools import reduce
reduce(function, iterable)
```

Example:

```python
reduce(lambda x, y: x * y, [1, 2, 3, 4])  # returns 24
```

Use to compute totals, combine results, and build accumulators.

`zip()`: Combine Iterables Element-wise

Pair elements from multiple iterables into tuples.

```python
names = ["Asha", "Ravi"]
scores = [90, 85]
print(list(zip(names, scores)))  # [('Asha', 90), ('Ravi', 85)]
```

Perfect for combining columns, features, or paired data.

`enumerate()`: Index While Iterating

Returns index-value pairs from an iterable.

```python
for idx, name in enumerate(["Asha", "Ravi"]):
    print(idx, name)
```

Useful in loops where you need both the item and position.

`any()` and `all()`: Boolean Checks

- `any()` → returns `True` if **any** item is true
- `all()` → returns `True` if **all** items are true

Example:

```python
marks = [70, 45, 30]
any(m < 40 for m in marks)  # True (failed at least once)
all(m >= 40 for m in marks)  # False
```

Used for validation, pass/fail checks, and filtering bad rows.

`sum()`, `min()`, `max()`: Aggregates

```python
nums = [10, 20, 30]
sum(nums)     # 60
min(nums)     # 10
max(nums)     # 30
```

Essential for summarising numeric datasets quickly.

`tqdm()`: Loop Progress Bars

Use `tqdm` to visualise progress for long loops or computations.

Installation:

```shell
pip install tqdm
```

Usage:

```python
from tqdm import tqdm
for i in tqdm(range(10000)):
    pass
```

Essential for ML training loops, data preprocessing, and simulations.

Ask an LLM

- How does `map()` compare to list comprehensions? Which is more Pythonic?
- Why might `reduce()` be considered less readable? When is it useful?
- What are some real-world examples of using `zip()` in DS or ML?
- Why use `enumerate()` instead of manually tracking an index variable?
- How does `tqdm` integrate with pandas or model training frameworks?

Here is a **short, focused lecture note on decorators** with relevant extensions like `@lru_cache`, context managers, and system hooks. The aim is to keep it practical and interview-aware without going deep into metaprogramming.

Decorators

A **decorator** is a function that **wraps another function** to add extra behaviour, without modifying its code.

Basic Example:

```python
def log(func):
    def wrapper(*args, **kwargs):
        print("Calling", func.__name__)
        return func(*args, **kwargs)
    return wrapper

@log
def greet(name):
    print(f"Hello {name}")
```

`@log` is shorthand for `greet = log(greet)`

Why Use Decorators?

- Add reusable behaviour (e.g. logging, timing, caching)
- Improve code modularity
- Used heavily in frameworks (Flask, TensorFlow, PySpark)

Useful Decorators

1. `@lru_cache` (from `functools`)

Caches the function output for faster repeated calls.

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    return n if n <= 1 else fib(n-1) + fib(n-2)
```

Use for recursive or expensive deterministic functions.

2. `@cache` (Python 3.9+)

Works like `@lru_cache(maxsize=None)` with simpler syntax.

3. Context Manager as Decorator

You can use `contextlib.contextmanager` to define setup/cleanup logic around code blocks.

```python
from contextlib import contextmanager

@contextmanager
def open_file(path):
     f = open(path, 'r')
     yield f
     f.close()
with open_file(filepath):
     pass
```

Enables use with `with` statement, common in file I/O and resource control.

4. `atexit.register`

Registers a function to run **when your script ends**.

```python
import atexit

@atexit.register
def goodbye():
    print("Cleaning up before exit...")
```

Use for saving logs, closing DBs, and flushing buffers.

5. Handling Interrupts with `signal`

Catch system signals like `Ctrl+C` (SIGINT):

```python
import signal
import sys

def handler(sig, frame):
    print("Interrupt received. Exiting safely.")
    sys.exit(0)

signal.signal(signal.SIGINT, handler)
```

Useful in long-running scripts, data processing pipelines.

Ask an LLM

- What does `@lru_cache` do internally?
- Why might you use a decorator in a ML pipeline?
- What's the difference between `with open()` and a custom context manager?
- How do `atexit` and `signal` help in production-grade scripts?

Try this out

1. Add a timer decorator that prints the execution time of any function.
2. Use `@lru_cache` to optimise a recursive factorial function.
3. Register a function with `atexit` to confirm graceful shutdown.
4. Simulate Ctrl+C and catch it with a signal handler.

Error Handling

- **Prevents crashes** in production pipelines
- Enables **clear messaging and graceful recovery**
- Allows **better debugging** and log tracing
- **Essential for interviews** and real-world utilities

The Basic Structure

```python
try:
    # code that might fail
except SomeError:
    # what to do if it fails
```

Example:

```python
try:
    x = int(input("Enter a number: "))
    print(100 / x)
except ValueError:
    print("Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

`else` and `finally` Blocks

- `else`: runs **only if no error occurs** in the `try` block
- `finally`: it always runs; useful for **cleanup code**

```python
try:

    # Code that may raise an error

except SomeError:

    # What to do if that error occurs

else:

    # What to do if no error occurs

finally:

    # Code that runs no matter what
```

Catching Multiple Errors

Use tuples to handle multiple exceptions in one block.

```python
try:
    risky_operation()
except (ValueError, TypeError) as e:
    print("Error occurred:", e)
```

Common Mistakes

Mistake	Why it's a problem
Catching `Exception` blindly	Mask unexpected issues
Empty `except:`	Silently swallows errors
Missing `finally` on the cleanup	Can leave resources open (e.g., files)

Real-World Patterns

File Handling:

```python
try:
    f = open("data.csv", "r")
    # process file
except FileNotFoundError:
    print("File not found.")
finally:
    f.close()
```

Input validation:

```python
def read_age():
    try:
        return int(input("Age: "))
    except ValueError:
        return 0
```

Logging Errors

Combine with `logging` to track errors instead of printing.

```python
import logging
logging.basicConfig(level=logging.WARNING)

try:
    x = 1 / 0
except ZeroDivisionError as e:
    logging.warning("Math failed: %s", e)
```

Logs are **preferable to print** in production or automation.

- Use `try/except` to handle predictable errors.
- Use `else` for clean runs, and `finally` for guaranteed cleanup.
- Avoid `except:` alone; always specify or log.
- For professional-grade scripts, **log exceptions instead of printing**.

Ask an LLM

- What happens if you raise an exception in `except`?
- What's the difference between `finally` and `except`?
- Why is it a bad idea to catch all exceptions blindly?
- How does structured error handling help in production systems?

Try this out

1. Wrap file I/O in a `try-except-finally` structure.
2. Build a function that logs every exception it encounters.
3. Simulate and handle a `TypeError` and `KeyError` in one block.

File Handling

Opening and Closing Files

You can open a file like this:

```python
file = open('data.txt', 'r')  # 'r' is for read mode
content = file.read()
print(content)
file.close
```

File Modes

Mode	Meaning	Behaviour
r	Read	Error if file doesn't exist
w	Write	Overwrites or creates a new file
a	Append	Adds to end; creates if not exists
x	Create	Fails if the file exists

Examples:

```python
f = open("new.txt", "w")
f.write("Hello World")

f = open("new.txt", "a") as f:
f.write("\nNew line added")
```

Opening Files with Context Managers

Use `with open(...) as ...` to handle file operations. It **automatically closes the file**, even if an error occurs.

```python
with open("data.txt", "r") as file:
    contents = file.read()
    print(contents)
```

Safer than `file = open(...); file.close()`

Working with File Paths (using `pathlib`)

Avoid OS-specific path issues using `pathlib`.

```python
from pathlib import Path

data_folder = Path("data")
file_path = data_folder / "users.json"

if file_path.exists():
    print(file_path.read_text())
```

Cross-platform, clean syntax

Error Handling with Files

Use `try-except` to catch file I/O issues:

```python
try:
    with open("missing.txt", "r") as f:
        data = f.read()
except FileNotFoundError:
    print("File not found.")
except PermissionError:
    print("Access denied.")
```

Reading & Writing Line-by-Line

```python
with open("log.txt") as f:
    for line in f:
        print(line.strip())

with open("log.txt", "w") as f:
    f.writelines(["line1\n", "line2\n"])
```

Useful for logs, structured text files, and batch processing

Working with JSON Files

Python provides a built-in `json` module.

Read JSON:

```python
import json

with open("data.json") as f:
    data = json.load(f)  # → dict/list
```

Write JSON:

```python
with open("output.json", "w") as f:
    json.dump(data, f, indent=2)
```

JSON from/to strings:

```python
json_str = json.dumps(data)
data = json.loads(json_str)
```

JSON is the go-to format for APIs, config files, and ML metadata

Try this out

1. Read a `.txt` file line by line and count the number of lines.
2. Write a Python list of dictionaries to a JSON file.
3. Handle a `FileNotFoundError` for a missing file, and create it instead.
4. Create a function `load_json(path)` that validates file existence and returns a Python object.
5. Use `pathlib` to check whether `logs/errors.log` exists and append to it safely.

Ask an LLM

- Why is `with open(...)` better than manually opening and closing a file?
- How would you handle corrupted or malformed JSON files?
- What are the differences between file modes `r+`, `w+`, and `a+`?
- How does `pathlib` improve portability?
- Why are context managers critical in production scripts?

Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a programming paradigm that organises code using **objects** — entities that bundle **data (attributes)** and **behaviour (methods)**.
It enables building modular, reusable, and maintainable code that mirrors real-world systems.

Why it matters:

- Encourages logical structuring and grouping of related functionality
- Widely used in data pipelines, simulation models, and machine learning APIs

Defining a Class and Creating Objects

A **class** is a blueprint for objects; an **object** is an instance of a class that contains actual data.

```python
class Person:
    def greet(self):
        print("Hello!")

# Creating an object
p = Person()
p.greet()
```

Each object created from a class has access to the methods defined inside the class.

Constructor: `__init__` Method

The `__init__` method is Python's constructor; it runs automatically when a new object is created.

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

It allows you to initialise attributes with custom values and ensures each object starts with a valid state.

Instance Methods and Variables

- **Instance variables** are defined within the `__init__` method and are unique to each object.
- **Instance methods** operate on those variables using the `self` keyword.

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

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

You can create multiple objects, each with its own values and method access.

Class Methods and Class Variables

- **Class variables** are shared across all instances of the class.
- **Class methods** (using `@classmethod`) can access and modify these shared variables via `cls`.

```python
class Car:
    wheels = 4

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

    @classmethod
    def describe(cls):
        return f"All cars have {cls.wheels} wheels."
```

Useful when maintaining state or behaviour that belongs to the class, not just individual objects.

Static Methods

- A `@staticmethod` doesn't access instance (`self`) or class (`cls`) data.
- It's a regular function logically grouped inside a class.

```python
class Math:
    @staticmethod
    def square(x):
        return x * x
```

Best used for helper functions, like validations, formatters, or computations.

Inheritance: Reuse and Extend

One class can **inherit** from another and reuse or extend its methods and attributes.

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

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

Inheritance supports **code reuse** and helps organise related classes in a hierarchy (e.g., `Vehicle → Car`).

Method Overriding

Child classes can **override** methods of their parent classes by redefining them.

```python
class Cat(Animal):
    def speak(self):
        print("Meow")
```

Use `super()` to call the parent method if needed. Overriding is essential for **customising inherited behaviour**.

upGrad

Core OOP Principles (The 4 Pillars)

Abstraction

- Expose only essential features; hide the internal complexity from the user
- Used in libraries like sklearn,` where models offer a simple interface like `.fit()` or `.predict()`

Encapsulation

- Python does not have *true* encapsulation; rather it relies on the developer to follow convention
- Bundle data and methods inside a class; restrict access using naming conventions (like `__variable`)
- This protects internal states from unintended modification and encourages safe access patterns

Python
```
class Account:
    def __init__(self):
        self.__balance = 0  # private variable

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

Inheritance

- Create child classes that inherit attributes and methods from a parent class.
- Enables hierarchical classification and eliminates code repetition.
- Used heavily in model abstraction: `BaseModel` → `LinearModel`, `TreeModel`, etc.

Polymorphism

- Write code that works on objects of different classes through a shared interface.
- Supports flexibility and clean design in APIs and utilities.

```python
for animal in [Dog(), Cat()]:
    animal.speak()  # Same method call → different output
```

Try this out

1. Build a `Book` class with title, author, and a method to display details.
2. Create a `Shape` base class and extend it to `Circle` and `Square` with overridden `area()` methods.
3. Use a class method to keep track of the number of `Employee` objects created.
4. Write a class with a static method to validate email addresses.
5. Demonstrate encapsulation using a class that prevents direct access to its balance.

Ask an LLM

- What's the difference between instance and class variables?
- When should you use a static method over a class method?
- How is encapsulation implemented in Python given that it doesn't enforce access modifiers?
- What role does polymorphism play in code extensibility?
- How does Python handle multiple inheritance?

Summary

Concept	Summary
`__init__`	The constructor is used for initialising the object state
`self`	Refers to the object instance
`@classmethod`	Works with class-level attributes via `cls`
`@staticmethod`	Independent utility method, logically grouped within the class
Inheritance	Reuse and extend functionality from base classes
Overriding	Redefine methods in derived classes
Abstraction	Hide implementation, expose interfaces
Encapsulation	Protect internal data, control access via methods
Polymorphism	Same interface, different class behaviours (dynamic method resolution)