Python

Basic — fundamentals you must know

1) **Hello, variables, types, printing**

```python
# variables and types
name = "Arman"
age = 30
height = 1.75
is_student = False

print("Name:", name)
print(f"{name} is {age} years old and {height}m tall.")
```

Explain: Python uses dynamic typing — variables are assigned without declaring types. f"..." are f-strings (formatted strings). print() outputs to console.

2) **Control flow — if / for / while**

```python
n = 5
if n % 2 == 0:
    print("even")
else:
    print("odd")

# for loop
for i in range(3):        # 0,1,2
    print(i)

# while loop
i = 0
while i < 3:
    print("while", i)
    i += 1
```

Explain: Indentation defines blocks. range(n) generates numbers; use break/continue as usual.

3) Functions & scope

```python
def greet(person: str) -> str:
    """Return a greeting for a person (type hints optional)."""
    return f"Hello, {person}!"

print(greet("Arman"))
```

Explain: def defines functions. Type hints (-> str) are optional and for readability / tooling.

3) Collections: list, tuple, dict, set

```python
# list (mutable)
fruits = ["apple", "banana"]
fruits.append("orange")

# tuple (immutable)
coord = (10, 20)

# dict (mapping)
student = {"name": "A", "age": 21}
print(student["name"])

# set (unique)
s = {1, 2, 2, 3}
print(s)  # {1,2,3}
```

Explain: Choose containers by mutability and access pattern.

5) Reading & writing files

```python
with open("sample.txt", "w", encoding="utf-8") as f:
    f.write("Hello file\n")

with open("sample.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

Explain: with ensures file is closed automatically (context manager).

Intermediate — practical patterns and idiomatic Python

1) List comprehensions & generator expressions

```python
nums = [1,2,3,4,5]
squares = [x*x for x in nums if x % 2 == 1]   # [1,9,25]
gen = (x*x for x in nums)                     # generator, lazy
```

Explain: Comprehensions are concise and fast. Generators produce values on demand.

2) Functions: default args, *args, **kwargs, docstrings

```python
def add(a, b=0):
    """Return a + b"""
    return a + b

def func(*args, **kwargs):
    print("positional:", args)
    print("keyword:", kwargs)

func(1,2, x=10, y=20)
```

Explain: *args collects positional extras as tuple; **kwargs collects keyword extras as dict. Avoid mutable default args (use None pattern).

2) Modules & packages

```python
# mymodule.py
def hello(): print("hello from module")

# usage
import mymodule
mymodule.hello()

# or
from mymodule import hello
hello()
```

Explain: Files are modules. Directories with __init__.py (optional in newer Python) are packages.

4) Exceptions and custom ones

```python
class MyError(Exception):
    pass

def divide(a, b):
    if b == 0:
        raise MyError("division by zero not allowed")
    return a / b

try:
    divide(1, 0)
except MyError as e:
    print("Caught:", e)
finally:
    print("cleanup")
```

Explain: Use exceptions for error handling. finally runs always.

5) File-like streaming, CSV example

```python
import csv

rows = [["name","age"], ["A", 21], ["B", 22]]
with open("out.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Explain: Python has batteries-included libs (csv, json, datetime, math...).

6) OOP basics: classes, inheritance, properties

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

    def speak(self):
        raise NotImplementedError

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

d = Dog("Rex")
print(d.speak())
```

Explain: self is instance. Use inheritance for shared behavior. Use @property for computed attributes if needed.

7) Iterators & generators (practical)

```python
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

for v in count_up_to(3):
    print(v)
```

Explain: yield creates a generator; useful for streaming large datasets.

8) Decorators (function wrappers)

```python
import time
def timeit(fn):
    def wrapper(*args, **kwargs):
        t0 = time.time()
        res = fn(*args, **kwargs)
        print(f"{fn.__name__} took {time.time()-t0:.6f}s")
        return res
    return wrapper

@timeit
def work(n):
    total = 0
    for i in range(n): total += i
    return total

work(100000)
```

Explain: Decorators wrap functions (e.g., logging, caching, access control).

Advanced — deep/modern Python features & patterns

1) Type hints + dataclasses

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

@dataclass
class Person:
    name: str
    age: int
    tags: List[str] = None

p = Person("Arman", 30, tags=["dev", "python"])
print(p)
```

Explain: Type hints help readability and linters. dataclass generates __init__, __repr__, etc.

2) Context managers (custom)

```python
from contextlib import contextmanager

@contextmanager
def open_transaction():
    print("BEGIN")
    try:
        yield
        print("COMMIT")
    except:
        print("ROLLBACK")
        raise

with open_transaction():
    print("do work")
    # raise Exception("oops")  # would trigger ROLLBACK
```

Explain: Context managers manage setup/teardown cleanly.

3) Async I/O (async/await)

```python
import asyncio

async def say_after(delay, msg):
    await asyncio.sleep(delay)
    print(msg)

async def main():
    await asyncio.gather(
        say_after(1, "hello"),
        say_after(2, "world"),
    )

asyncio.run(main())
```

Explain: asyncio for concurrent I/O tasks (network, file I/O with async libs). Not for CPU-bound work — use threads/processes.

4) Concurrency: threads vs processes

```python
# CPU-bound => multiprocessing
from concurrent.futures import ProcessPoolExecutor

def heavy(x):
    return x*x

with ProcessPoolExecutor() as ex:
    print(list(ex.map(heavy, range(5))))
```

Explain: Use threads for I/O-bound tasks, processes for CPU-bound tasks (GIL restricts threads).

5) Metaprogramming: dynamic attributes, metaclasses (short)

```
# dynamic attribute injection
class A: pass
a = A()
setattr(a, "x", 10)
print(a.x)

# metaclass example (advanced)
class VerboseMeta(type):
    def __new__(mcls, name, bases, namespace):
        print("Creating", name)
        return super().__new__(mcls, name, bases, namespace)

class My(metaclass=VerboseMeta):
    pass
```

Explain: Metaclasses control class creation — used rarely (frameworks, ORMs).

---

6) Performance tips & memory

- Prefer list comprehensions over manual loops for speed.
- Use generators to avoid memory spikes for large sequences.
- Use __slots__ in classes if you need many instances and want lower memory overhead.
- Profile with cProfile / timeit before optimizing.

---

7) Packaging & testing (short)

- Use pytest for tests; write small, isolated tests.
- Package code with pyproject.toml (PEP 517/518) and publish to PyPI if needed.

Guided mini-project ideas to practice

- Basic: CLI TODO app (file-based).
- Intermediate: Web scraper that saves JSON (use requests, BeautifulSoup).
- Advanced: Async web crawler or REST API with FastAPI + background workers + type hints + tests.

Quick style & tooling tips

- Follow PEP8 (use black for formatting).
- Use virtual environments (python -m venv .venv) or poetry.
- Add types gradually — they help maintainability.
- Write tests early.