Basic Concepts

Got it 👍 — here's a breakdown of the **important concepts in Python** that you should understand to become proficient:

🔷 1. **Basics**

- **Variables & Data Types**: `int`, `float`, `str`, `bool`, `list`, `tuple`, `set`, `dict`
- **Operators**: arithmetic, comparison, logical, bitwise, assignment
- **Type Conversion**: implicit vs explicit (`int()`, `float()`, `str()`)

🔷 2. **Control Flow**

- **Conditionals**: `if`, `elif`, `else`
- **Loops**: `for`, `while`
- **Loop Controls**: `break`, `continue`, `pass`

🔷 3. **Functions**

- Defining with `def`
- Arguments & Parameters:
  - positional, keyword, default, variable-length (`args`, `*kwargs`)
- Return values
- Lambda functions (`lambda`)
- Scope & lifetime of variables (local vs global)

🔷 4. **Data Structures**

- **Lists**: mutable sequences
- **Tuples**: immutable sequences

- **Sets**: unique, unordered items
- **Dictionaries**: key-value pairs
- List comprehensions, set/dict comprehensions

🔷 5. **Modules & Packages**

- Importing (`import`, `from ... import ...`)
- Standard library (e.g., `math`, `datetime`, `random`)
- Creating custom modules
- `pip` for external packages

🔷 6. **Object-Oriented Programming (OOP)**

- Classes & Objects
- `__init__` constructor
- Instance vs class variables
- Methods (instance, class, static)
- Inheritance & Polymorphism
- Encapsulation (`private`, `protected`, `public`)
- Special methods (`__str__`, `__len__`, `__repr__`)

🔷 7. **Exception Handling**

- `try`, `except`, `finally`
- Raising exceptions (`raise`)
- Custom exceptions

🔷 8. **File Handling**

- Reading & writing (`open`, `with`)
- Modes: `r`, `w`, `a`, `b`

- File paths & context managers

🔷 9. **Advanced Concepts**

- Iterators & Generators (`yield`)
- Decorators
- Context Managers (`with`)
- Regular Expressions (`re` module)
- Virtual environments (`venv`, `conda`)

🔷 10. **Pythonic Practices**

- List comprehensions
- `enumerate()` and `zip()`
- `map()`, `filter()`, `reduce()`
- F-strings (`f"Hello {name}"`)
- `args` and `*kwargs`

⚡ Pro Tip: Learn these in **layers** — start with **basics + control flow**, then move to **functions + data structures**, and finally dive into **OOP + advanced concepts** when you're comfortable.

Would you like me to make you a **roadmap (step-by-step order)** for learning these concepts, or do you want me to **explain each with examples**?

🟦 Step 1 — Basics

Concepts (super short)

- **Types**: `int`, `float`, `str`, `bool`, `list`, `tuple`, `set`, `dict`
- **Operators**: arithmetic `+ - * / // % **`, comparison, logical `and or not`

- **Type casting**: `int()`, `float()`, `str()`, `bool()`
- **Input/Output**: `input()`, `print()`
- **Slicing**: `seq[start:stop:step]`

Exercises + Solutions

Beginner

```
# 1 type check
a, b, c, d = 5, 3.14, "hi", True
print(type(a), type(b), type(c), type(d))
```

```
# 2 integer division vs modulus
x = 17
print(x // 5, x % 5)  # 3 2
```

```
# 3 casting
s = "42"
print(int(s) + 8)  # 50
```

```
# 4 string slicing
msg = "python"
print(msg[:3], msg[-3:])  # pyt hon
```

Intermediate

```
# 5 f-strings
name, score = "Ava", 93
print(f"{name} scored {score}/100")
```

```
# 6 unpacking
a, b, *rest = [10, 20, 30, 40]
```

```
print(a, b, rest)  # 10 20 [30, 40]
```

```
# 7 truthy/falsey
values = [0, "", [], [1], "ok"]
print([bool(v) for v in values])  # [False, False, False, True, True]
```

Advanced

```
# 8 chained comparison
x = 7
print(5 < x < 10)  # True
```

```
# 9 immutability demo
t = (1,2,3)
# t[0] = 9  # TypeError
```

```
# 10 dict basics
person = {"name":"Kai","age":22}
print(person.get("city","N/A"))
```

🟧 Step 2 — Control Flow

Concepts

- **Conditionals**: `if / elif / else`
- **Loops**: `for` (iterate over items), `while` (repeat until condition)
- **Loop controls**: `break`, `continue`, `pass`
- **Range**: `range(start, stop, step)`
- **Ternary**: `a if cond else b`

Exercises + Solutions

Beginner

```
# 1 basic if-elif-else
n = 0
if n > 0: print("pos")
elif n < 0: print("neg")
else: print("zero")
```

```
# 2 for loop sum 1..10
total = 0
for i in range(1, 11):
    total += i
print(total)  # 55
```

```
# 3 while countdown
i = 5
while i > 0:
    print(i)
    i -= 1
```

Intermediate

```
# 4 break on first multiple of 7
for i in range(1, 100):
    if i % 7 == 0:
        print(i)
        break
```

```
# 5 continue: print non-multiples of 3 up to 10
for i in range(1, 11):
    if i % 3 == 0:
```

```
        continue
    print(i, end=" ")
# 1 2 4 5 7 8 10
```

```
# 6 ternary
age = 17
status = "adult" if age >= 18 else "minor"
print(status)
```

Advanced

```
# 7 nested loops: multiplication table (1..3)
table = []
for i in range(1,4):
    row = []
    for j in range(1,4):
        row.append(i*j)
    table.append(row)
print(table)  # [[1,2,3],[2,4,6],[3,6,9]]
```

```
# 8 for-else: check prime (else runs if no break)
n = 13
is_prime = True
for d in range(2, int(n**0.5)+1):
    if n % d == 0:
        is_prime = False
        break
else:
    is_prime = True
print(is_prime)  # True
```

```
# 9 enumerate + break
names = ["Ana","Ben","Cara"]
```

```
for i, nm in enumerate(names):
    if nm == "Ben":
        print("found at", i)
        break
```

🟩 Step 3 — Functions

Concepts

- Define with `def`; return with `return`
- Parameters: positional, keyword, default, `args`, `*kwargs`
- Scope: local vs global; avoid `global` unless needed
- First-class functions: pass them around
- Lambda: small anonymous function
- Docstrings: `"""describe function"""`

Exercises + Solutions

Beginner

```
# 1 simple add
def add(a, b):
    return a + b
print(add(2, 3))
```

```
# 2 default argument
def greet(name="there"):
    return f"Hello, {name}!"
print(greet(), greet("Sam"))
```

```
# 3 docstring + help
def area_circle(r):
    """Return area of a circle of radius r."""
    from math import pi
    return pi * r * r

# print(area_circle.__doc__)
```

Intermediate

```
# 4 *args sum
def total(*nums):
    return sum(nums)
print(total(1,2,3))  # 6
```

```
# 5 **kwargs config
def connect(**cfg):
    host = cfg.get("host","localhost")
    port = cfg.get("port", 5432)
    return f"Connecting to {host}:{port}"
print(connect(host="db.local", port=3306))
```

```
# 6 keyword-only & positional-only (3.8+)
def rect_area(width, /, *, height):
    return width * height
print(rect_area(5, height=4))  # 20
```

```
# 7 lambda + sort by length
words = ["pear","banana","fig","apple"]
words.sort(key=lambda w: len(w))
print(words)  # ['fig', 'pear', 'apple', 'banana']
```

Advanced

```
# 8 higher-order function
def apply_twice(fn, x):
    return fn(fn(x))

def inc(n): return n+1
print(apply_twice(inc, 3))  # 5
```

```
# 9 closures (make multiplier)
def make_mul(k):
    def mul(x): return k * x
    return mul

double = make_mul(2)
print(double(7))  # 14
```

```
# 10 recursion (factorial)
def fact(n):
    return 1 if n <= 1 else n * fact(n-1)
print(fact(5))  # 120
```

```
# 11 argument unpacking
def trio(a,b,c): return a+b+c
nums = [1,2,3]
print(trio(*nums))  # 6
```

```
# 12 annotations (type hints)
def greet_user(name: str, excited: bool = False) -> str:
    return f"Hello, {name}{'!' if excited else '.'}"
```

🟦 Step 4: Python Data Structures

🔷 1. **Lists (beginner → intermediate)**

- Ordered, mutable (changeable).
- Can store mixed data types.

```
# Create list
fruits = ["apple", "banana", "cherry"]

# Access
print(fruits[0])     # apple
print(fruits[-1])    # cherry

# Modify
fruits[1] = "blueberry"

# Add / Remove
fruits.append("orange")    # add at end
fruits.insert(1, "kiwi")   # add at index
fruits.remove("apple")     # remove by value
fruits.pop()               # remove last
```

👉 Useful methods: `.sort()`, `.reverse()`, `.count()`, `.extend()`

✅ **Intermediate – List comprehension**

```
squares = [x**2 for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]
```

🔷 2. **Tuples (beginner)**

- Ordered, **immutable** (cannot be changed after creation).

- Faster than lists.

```
coords = (10, 20)
print(coords[0])    # 10

# Single element tuple
one_item = (5,)   # must add comma
```

👉 Often used for fixed data (e.g., `(latitude, longitude)`).

🔷 3. Sets (beginner → intermediate)

- **Unordered, unique elements**, mutable.
- Automatically removes duplicates.

```
nums = {1, 2, 3, 3, 2}
print(nums)   # {1, 2, 3}

# Add / Remove
nums.add(4)
nums.remove(2)

# Set operations
a = {1, 2, 3}
b = {3, 4, 5}
print(a | b)   # Union → {1, 2, 3, 4, 5}
print(a & b)   # Intersection → {3}
print(a - b)   # Difference → {1, 2}
```

👉 Great for membership tests (`if x in set`) because lookup is **O(1)**.

🔷 4. Dictionaries (intermediate → advanced)

- Key-Value pairs.
- Keys must be **immutable** (`str`, `int`, `tuple`), values can be anything.

```
person = {"name": "Alice", "age": 25}

# Access
print(person["name"])      # Alice
print(person.get("height", "N/A"))  # safer access

# Add / Update
person["city"] = "New York"
person["age"] = 26

# Delete
del person["city"]
```

👉 Iteration:

```
for key, value in person.items():
    print(key, value)
```

👉 Dictionary comprehension:

```
squares = {x: x**2 for x in range(5)}
print(squares)  # {0:0, 1:1, 2:4, 3:9, 4:16}
```

🔷 5. Advanced Pythonic Data Structures

a) Nested Structures

```
students = [
   {"name": "Alice", "marks": [90, 85, 92]},
   {"name": "Bob", "marks": [70, 88, 75]}
]
print(students[0]["marks"][1])  # 85
```

b) Collections Module

Python gives you **specialized structures**:

```
from collections import deque, Counter, defaultdict

# deque → fast appends and pops
queue = deque([1,2,3])
queue.appendleft(0)   # [0,1,2,3]

# Counter → frequency count
cnt = Counter("banana")
print(cnt)   # {'a': 3, 'n': 2, 'b': 1}

# defaultdict → default value for missing keys
dd = defaultdict(int)
dd["apple"] += 1
print(dd["apple"], dd["banana"])  # 1, 0
```

c) Sets vs Frozensets

- `set` = mutable
- `frozenset` = immutable, can be dictionary keys.

📝 Data Structures Practice Exercises

🔷 1. Lists

Beginner

1. Create a list of 5 numbers and print the 2nd and last element.
2. Replace the 3rd element with `99`.
3. Add `"hello"` at the end of the list.
4. Remove the first element.

Intermediate

5. Create a list of 10 numbers. Print only the even ones using slicing or loops.
6. Use **list comprehension** to make a list of squares from `1–10`.
7. Reverse a list without using `.reverse()` or slicing shortcut.

Advanced

8. Flatten this list into a single list: `[[1,2],[3,4],[5,6]] → [1,2,3,4,5,6]`.
9. Remove duplicates from a list while preserving order.
10. Rotate a list by 2 steps: `[1,2,3,4,5] → [4,5,1,2,3]`.

🔷 2. Tuples

Beginner

1. Create a tuple with 4 elements.
2. Try changing one element → note the error.
3. Convert the tuple to a list, modify it, then convert back to a tuple.

Intermediate

4. Unpack the tuple `(1, 2, 3)` into variables `a, b, c`.
5. Swap two numbers using a tuple (no temp variable).

Advanced

6. Create a function that returns multiple values as a tuple (e.g., min and max of a list).

🔷 3. Sets

Beginner

1. Create a set of numbers with duplicates and print it (to see removal of duplicates).
2. Add and remove elements from the set.
3. Check if a number exists in the set.

Intermediate

4. Perform union, intersection, and difference on two sets `{1,2,3}` and `{3,4,5}`.
5. Find all unique characters in the string `"mississippi"`.

Advanced

6. Given a list of numbers, find which numbers appear more than once using sets.

🔷 4. Dictionaries

Beginner

1. Create a dictionary with keys: `"name"`, `"age"`, `"city"`.
2. Access and print `"name"`.
3. Update `"age"` to 30.
4. Delete `"city"`.

Intermediate

5. Loop through dictionary and print key-value pairs.
6. Create a dictionary of squares: `{1:1, 2:4, 3:9, ..., 10:100}`.
7. Given two lists `keys = ["a","b","c"]` and `values = [1,2,3]`, create a dictionary.

Advanced

8. Count the frequency of each word in a sentence using a dictionary.

9. Create a nested dictionary for students and their subjects:

```
{
   "Alice": {"math": 90, "science": 85},
   "Bob": {"math": 70, "science": 88}
}
```

Access Bob's science mark.

10. Merge two dictionaries into one without using `{**dict1, **dict2}`.

👉 Suggestion: Start with **Lists (Beginner)** and show me your answers.

I'll check them and guide you before moving to the next level.

Do you want me to give you **solutions after each set**, or keep them hidden so you can try first?

You got it! Here are clean, idiomatic solutions for every exercise. Try running them one-by-one and tweak to see variations.

✅ Solutions

1) Lists

Beginner

```
# 1
nums = [10, 20, 30, 40, 50]
print(nums[1], nums[-1])  # 20 50

# 2
nums[2] = 99  # [10, 20, 99, 40, 50]

# 3
```

```
nums.append("hello")  # [10, 20, 99, 40, 50, 'hello']

# 4
del nums[0]  # [20, 99, 40, 50, 'hello']
```

Intermediate

```
# 5
lst = list(range(1, 11))
evens = [x for x in lst if x % 2 == 0]
print(evens)  # [2,4,6,8,10]

# 6
squares = [x*x for x in range(1, 11)]
print(squares)

# 7 (reverse without .reverse() or slicing)
lst = [1, 2, 3, 4, 5]
rev = []
for i in range(len(lst)-1, -1, -1):
    rev.append(lst[i])
print(rev)  # [5,4,3,2,1]
```

Advanced

```
# 8 Flatten
nested = [[1,2],[3,4],[5,6]]
flat = [x for sub in nested for x in sub]
print(flat)  # [1,2,3,4,5,6]

# 9 Remove duplicates preserving order
data = [3,1,2,3,2,4,1,5]
seen = set()
dedup = [x for x in data if not (x in seen or seen.add(x))]
print(dedup)  # [3,1,2,4,5]
```

```
# 10 Rotate by 2 (right-rotate)
arr = [1,2,3,4,5]
k = 2 % len(arr)
rotated = arr[-k:] + arr[:-k]
print(rotated)  # [4,5,1,2,3]
```

---

2) Tuples

Beginner

```
# 1
t = (10, 20, 30, 40)

# 2 (will raise TypeError if you try)
# t[1] = 99  # TypeError: 'tuple' object does not support item assignment

# 3
lst = list(t)
lst[1] = 99
t2 = tuple(lst)
print(t2)  # (10, 99, 30, 40)
```

Intermediate

```
# 4
a, b, c = (1, 2, 3)

# 5 Swap without temp
x, y = 10, 20
x, y = y, x  # x=20, y=10
```

Advanced

```
# 6 Function returning multiple values (min, max)
def min_max(seq):
    mn = min(seq)
    mx = max(seq)
    return mn, mx

print(min_max([5, 2, 9, 1]))  # (1, 9)
```

---

3) Sets

Beginner

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

# 2
s.add(4)
s.remove(2)  # or s.discard(2) to avoid KeyError
print(s)

# 3
print(3 in s)  # True/False
```

Intermediate

```
# 4
a, b = {1,2,3}, {3,4,5}
print(a | b)  # union
print(a & b)  # intersection
print(a - b)  # difference

# 5
```

```
unique_chars = set("mississippi")
print(unique_chars)  # {'i','m','p','s'}
```

Advanced

```
# 6 numbers that appear more than once
nums = [1,2,3,2,4,1,5,1]
seen, dupes = set(), set()
for n in nums:
    if n in seen:
        dupes.add(n)
    else:
        seen.add(n)
print(dupes)  # {1,2}
```

---

4) Dictionaries

Beginner

```
# 1
person = {"name": "Alice", "age": 25, "city": "NYC"}

# 2
print(person["name"])  # Alice

# 3
person["age"] = 30

# 4
del person["city"]
```

Intermediate

```
# 5
for k, v in person.items():
    print(k, v)

# 6
squares = {x: x*x for x in range(1, 11)}
print(squares)

# 7
keys = ["a","b","c"]
values = [1,2,3]
d = dict(zip(keys, values))
print(d)  # {'a':1, 'b':2, 'c':3}
```

Advanced

```
# 8 word frequency
sentence = "to be or not to be that is the question"
freq = {}
for word in sentence.split():
    freq[word] = freq.get(word, 0) + 1
print(freq)

# 9 nested dict access
students = {
    "Alice": {"math": 90, "science": 85},
    "Bob": {"math": 70, "science": 88}
}
print(students["Bob"]["science"])  # 88

# 10 merge two dicts without {**a, **b}
d1 = {"x":1, "y":2}
d2 = {"y":99, "z":3}
merged = d1.copy()
```

```
merged.update(d2)
print(merged)  # {'x':1, 'y':99, 'z':3}
```

🟦 Step 5: Modules & Packages

🔷 1. What is a Module?

- A **module** is just a `.py` file with Python code (functions, classes, variables).
- Helps organize code into separate files.

Example

```
# math is a built-in module
import math

print(math.sqrt(16))   # 4.0
print(math.pi)         # 3.14159...
```

👉 Aliases:

```
import math as m
print(m.factorial(5))   # 120
```

👉 Import specific things:

```
from math import sqrt, pi
print(sqrt(25), pi)
```

🔷 2. Standard Library (batteries included 🪫🔋)

Python comes with **tons of built-in modules**. Some popular ones:

- `random` → randomness

- `datetime` → dates & times
- `os` / `sys` → operating system tasks
- `json` → JSON data handling
- `re` → regex
- `collections` → advanced data structures

Example

```
import random, datetime

print(random.randint(1, 100))   # random number
print(datetime.datetime.now())  # current date/time
```

🔷 3. Packages

- A **package** = folder with multiple modules and an `__init__.py` file.
- Example: `numpy`, `pandas` are **packages**.
- Installed via **pip**:

```
pip install requests
```

Example (using requests)

```
import requests

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.json())
```

🔷 4. Creating Your Own Module

Say you make a file `mymath.py`:

```
# mymath.py
def add(a, b):
    return a + b
```

Use it in another file:

```
import mymath

print(mymath.add(5, 3))  # 8
```

📝 Exercises

Beginner

1. Import `math` and print:
    - square root of 64
    - factorial of 6
    - value of pi
2. Use `random` to generate:
    - a random integer from 1–10
    - a random element from `["apple", "banana", "cherry"]`

Intermediate

1. Use `datetime` to:
    - print today's date
    - print current year, month, and day separately
2. Use `os` to:
    - print current working directory
    - list files in it

Advanced

1. Use `collections.Counter` to count frequency of characters in `"hello world"`.
2. Create a **custom module** `utils.py` with:
   - `is_even(num)` → returns True/False
   - `reverse_string(s)` → returns reversed string

   Import and use it in another script.

🟦 Step 5: Modules & Packages — Solutions

Beginner

```
# 1
import math
print(math.sqrt(64))   # 8.0
print(math.factorial(6))  # 720
print(math.pi)  # 3.141592653589793

# 2
import random
print(random.randint(1, 10))
print(random.choice(["apple", "banana", "cherry"]))
```

Intermediate

```
# 3
import datetime as dt
today = dt.date.today()
print(today)
print(today.year, today.month, today.day)
```

```
# 4
import os
print(os.getcwd())
print(os.listdir(os.getcwd()))
```

Advanced

```
# 5
from collections import Counter
print(Counter("hello world"))

# 6 (create utils.py in same folder)
# --- utils.py ---
def is_even(num): return num % 2 == 0
def reverse_string(s): return s[::-1]

# --- main.py ---
import utils
print(utils.is_even(42))       # True
print(utils.reverse_string("abc"))  # cba
```

🟧 Step 6: Object-Oriented Programming (OOP)

Core ideas

- **Class** = blueprint; **Object** = instance.
- `__init__` constructor, `self` for instance data.
- Methods: instance, `@classmethod`, `@staticmethod`.
- Inheritance, overriding, `super()`.
- Dunder methods (`__str__`, `__len__`, `__eq__`, ...).

Exercises + Solutions

Beginner

```
# 1 Define a class with attributes + method
class Dog:
    def __init__(self, name, age):  # constructor
        self.name = name
        self.age = age
    def bark(self):
        return f"{self.name} says woof!"

d = Dog("Rex", 3)
print(d.bark())  # Rex says woof!
```

```
# 2 Add a method to compute "human years" (age*7)
class Dog:
    def __init__(self, name, age):
        self.name, self.age = name, age
    def human_years(self):
        return self.age * 7

print(Dog("Milo", 4).human_years())  # 28
```

Intermediate

```
# 3 Class vs static vs instance methods
class MathBox:
    scale = 10  # class attribute

    def __init__(self, x):      # instance method
        self.x = x

    def scaled(self):
        return self.x * MathBox.scale
```

```
    @classmethod
    def set_scale(cls, value):  # class method
        cls.scale = value

    @staticmethod
    def is_positive(n):         # static method
        return n > 0

m = MathBox(5)
print(m.scaled())         # 50
MathBox.set_scale(3)
print(m.scaled())         # 15
print(MathBox.is_positive(-1))  # False
```

```
# 4 Inheritance + method override
class Shape:
    def area(self): raise NotImplementedError

class Rectangle(Shape):
    def __init__(self, w, h): self.w, self.h = w, h
    def area(self): return self.w * self.h

class Square(Rectangle):
    def __init__(self, s):
        super().__init__(s, s)

print(Rectangle(3,4).area())  # 12
print(Square(5).area())       # 25
```

Advanced

```
# 5 Dunder methods for nice behavior
class Vector2D:
    def __init__(self, x, y): self.x, self.y = x, y
    def __add__(self, other): return Vector2D(self.x+other.x, self.y+other.y)
```

```
    def __eq__(self, other): return (self.x, self.y) == (other.x, other.y)
    def __repr__(self): return f"Vector2D({self.x}, {self.y})"
    def __len__(self): return 2

a, b = Vector2D(1,2), Vector2D(3,4)
print(a + b)       # Vector2D(4, 6)
print(a == Vector2D(1,2))  # True
print(len(a))      # 2
```

🟥 Step 7: Exception Handling

Core ideas

- Use `try/except/else/finally`.
- `raise` to throw; custom exception classes.
- Catch specific exceptions, not a blanket `except`.

Exercises + Solutions

Beginner

```
# 1 Handle ZeroDivisionError
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

```
# 2 Safe int parsing
def to_int(s):
    try:
        return int(s)
    except ValueError:
        return None
```

```
print(to_int("42")), print(to_int("x"))  # 42, None
```

Intermediate

```
# 3 Multiple except + else + finally
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "division by zero"
    else:
        return result
    finally:
        pass  # cleanup/logging place

print(divide(10, 2))  # 5.0
```

Advanced

```
# 4 Custom exception
class NegativeAmountError(Exception): pass

def withdraw(balance, amount):
    if amount < 0:
        raise NegativeAmountError("Amount cannot be negative")
    if amount > balance:
        raise ValueError("Insufficient funds")
    return balance - amount

try:
    print(withdraw(100, 150))
except ValueError as e:
    print(e)
```

🟩 Step 8: File Handling

Core ideas

- Use `with open(path, mode)` to auto-close.
- Modes: `r`, `w`, `a`, `b`, `x`.
- Read: `.read()`, `.readline()`, `.readlines()`. Write: `.write()`.

Exercises + Solutions

Beginner

```
# 1 Write a file
with open("notes.txt", "w", encoding="utf-8") as f:
    f.write("Hello\nWorld")

# 2 Read it back line by line
with open("notes.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

Intermediate

```
# 3 Copy a text file
with open("notes.txt", "r", encoding="utf-8") as src, open("copy.txt", "w", enc
oding="utf-8") as dst:
    for chunk in src:
        dst.write(chunk)
```

```
# 4 JSON read/write
import json
data = {"name":"Alice","age":25,"skills":["py","sql"]}
with open("user.json","w",encoding="utf-8") as f:
    json.dump(data, f, indent=2)
```

```
with open("user.json","r",encoding="utf-8") as f:
    loaded = json.load(f)
print(loaded["skills"][0])  # py
```

Advanced

```
# 5 CSV processing
import csv
rows = [
    ["name","score"],
    ["Alice", 90],
    ["Bob", 78],
]
with open("scores.csv","w",newline="",encoding="utf-8") as f:
    csv.writer(f).writerows(rows)

# Sum scores
total = 0
with open("scores.csv","r",encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for r in reader:
        total += int(r["score"])
print(total)  # 168
```

🟪 Step 9: Advanced Concepts

9.1 Iterators & Generators

- **Iterator**: object with `__iter__()` and `__next__()`.
- **Generator**: function using `yield` to lazily produce values.

Exercises + Solutions

```
# A) Generator of Fibonacci (n terms)
def fib(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a+b

print(list(fib(7)))  # [0,1,1,2,3,5,8]
```

```
# B) Custom iterator counting down
class Countdown:
    def __init__(self, n): self.n = n
    def __iter__(self): return self
    def __next__(self):
        if self.n <= 0: raise StopIteration
        self.n -= 1
        return self.n + 1

print(list(Countdown(5)))  # [5,4,3,2,1]
```

9.2 Decorators

- Wrap a function to add behavior.

```
# C) Timing decorator
import time
def timing(fn):
    def wrapper(*args, **kwargs):
        start = time.time()
        try:
            return fn(*args, **kwargs)
        finally:
            print(f"{fn.__name__} took {time.time()-start:.6f}s")
    return wrapper
```

```
@timing
def work(n): return sum(range(n))
print(work(10_0000))
```

9.3 Context Managers

- Use `with` to manage setup/teardown. Create via class or `contextlib`.

```
# D) Simple timer context manager
import time
from contextlib import contextmanager

@contextmanager
def timer():
    t0 = time.time()
    try:
        yield
    finally:
        print(f"Elapsed: {time.time()-t0:.6f}s")

with timer():
    sum(range(1_000_00))
```

9.4 Regular Expressions

```
# E) Extract emails from text
import re
text = "Contact us: a@x.com, b.y@z.org; or visit site"
emails = re.findall(r"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}", text)
print(emails)
```

9.5 Virtual Environments (concept)

```
# F) (terminal)
python -m venv .venv
# activate:
# Windows: .venv\Scripts\activate
# macOS/Linux: source .venv/bin/activate
pip install requests
```

🟨 Step 10: Pythonic Practices

Concepts

- Comprehensions, unpacking, EAFP (“ask forgiveness”), `enumerate`, `zip`, f-strings, truthy/falsey, packing `args/**kwargs`.

Exercises + Solutions

```
# 1 List/Dict comprehensions
squares = [x*x for x in range(10) if x%2==0]
index_map = {c:i for i, c in enumerate("python")}
print(squares, index_map)
```

```
# 2 enumerate + zip
names = ["Alice","Bob","Cara"]
scores = [90, 78, 88]
for i, (n, s) in enumerate(zip(names, scores), start=1):
    print(f"{i}. {n}: {s}")
```

```
# 3 EAFP style dict access with defaults
user = {"name":"Kai"}
age = user.get("age", 0)
print(age)  # 0
```

```
# 4 Unpacking & argument forwarding
def add(a, b, c): return a+b+c
nums = (1,2,3)
print(add(*nums))  # 6

def call_twice(fn, *args, **kwargs):
    return fn(*args, **kwargs), fn(*args, **kwargs)

print(call_twice(max, [1,9,3]))
```

```
# 5 F-strings
name, score = "Dana", 95
print(f"{name} scored {score}/100")
```

What’s next?

If you want to **apply all of this**, I can give you a compact **mini-project** (CLI to manage tasks: add/list/complete, with file persistence, OOP, exceptions, and tests), with **step-by-step tasks + solutions**. Want that?

🧩 Project: CLI Task Manager