PYTHON INTERMEDIATE

BY COMPUTER CLUB KMUTNB

ERROR HANDLING

WHY AND WHEN ERROR HANDLING?

OVERVIEW

Common Use Cases:
- User input errors (e.g., int("abc"))
- Division by zero
- Missing or unexpected data in a variable

Use error handling when:
- You can't trust the input
- Something might behave unpredictably
- You want the program to survive the error

Try to “catch” problems instead of crashing

DIFFERENCE BETWEEN EXCEPTION AND ERROR

OVERVIEW

Exceptions are problems your program can expect and handle, such as ValueError, ZeroDivisionError, or TypeError

Errors are more serious issues that usually mean something is wrong with the program itself, such as MemoryError, SyntaxError, or SystemError

WITHOUT ERROR HANDLING

ERROR HANDLING

If the user types ‘abc’ into the following code

```
1 age = int(input("Enter your age: "))
```

Python Output:

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    age = int(input("Enter your age: "))
ValueError: invalid literal for int() with base 10: 'abc'
```

This error is caused by trying to convert text that isn’t a number (like "abc") into an integer using int()

COMMON EXCEPTIONS IN PYTHON

ERROR HANDLING

Exception	When it Happens
ValueError	When a value is the right type but invalid (e.g., `int("abc")`)
TypeError	When an operation is used on the wrong type (e.g., adding string to int)
ZeroDivisionError	When you divide by zero (`10 / 0`)
IndexError	When you access a list index that doesn't exist
KeyError	When a dictionary key is missing
FileNotFoundError	When trying to open a file that doesn’t exist
NameError	When using a variable that hasn’t been defined

HANDLING A SIMPLE EXCEPTION

ERROR HANDLING

If the user types ‘abc’ into the following code

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a number.")
```

Python Output:

```
Please enter a number.
```

Now, instead of throwing a ValueError and crashing the program, Python catches the error and prints a friendly message

HANDLING MULTIPLE EXCEPTIONS

ERROR HANDLING

You can use multiple except blocks to handle different kinds of errors separately

```python
try:
    num = int("0")
    result = 10 / num
except ValueError:
    print("Not an integer.")
except ZeroDivisionError:
    print("Can't divide by zero.")
```

Python Output:

```
Can't divide by zero.
```

The string "0" is successfully converted to the number 0, but dividing by zero causes a ZeroDivisionError, so the program prints “Can't divide by zero.” instead of crashing

HANDLING WITH ELSE AND FINALLY

ERROR HANDLING

```python
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That’s not a valid number.")
else:
    print("Great! You entered:", number)
finally:
    print("This runs no matter what.")
```

try: Attempts to convert the input to an integer
except: If input is invalid (e.g., "abc"), prints an error message
else: If the input is valid, prints the number
finally: Always runs at the end (used here for a closing message)

CATCHES ALL EXCEPTIONS

ERROR HANDLING

```python
try:
    risky_code()
except:
    print("Something went wrong.")
```

This catches all exceptions — even ones you didn’t expect

Why it’s risky to catches all exceptions?
Catching all exceptions hides the real problem, makes debugging harder, and may catch errors you didn’t intend to handle

CAPTURE THE ERROR MESSAGE

ERROR HANDLING

```python
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative.")
except ValueError as e:
    print("Error:", e)
```

as e lets you capture the error message into a variable (e)
You can then print or log the exact reason the error happened

RAISE AN EXCEPTION

ERROR HANDLING

```python
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative.")
except ValueError as e:
    print("Error:", e)
else:
    print("Thank you! Your age is:", age)
```

Normally, this wouldn’t raise an exception — but with raise, we force the program to stop and throw an error
This is useful for custom exceptions or handling foreseen problems like invalid input

FILE HANDLING

WHAT IS FILE HANDLING?

OVERVIEW

File handling is how Python lets you open, read, write, and update files stored on your computer

Why we use it:

- To load data from a file (like .txt or .json)
- To save data that your code creates
- To preserve information between runs of your program

TYPES OF FILES

OVERVIEW

File Type	Extension	What It Looks Like	What It's Good For
Text File	.txt	Hello, world!``This is a note.	Notes, logs, simple line-based data
JSON File	.json	{ "name": "Alice", "age": 30 } [{"a": 1}, {"b": 2}]	Structured data like lists & dictionaries (like mini databases!)

MANUAL FILE HANDLING

FILE HANDLING

```python
try:
    file = open('message.txt', 'r')
    content = file.read()
    print(content)
finally:
    file.close()
```

open('message.txt', 'r')

- Opens the file named message.txt
- 'r' means read mode
- Returns a file object (like a “connection” to the file)

file.close()

- Closes the file manually
- 🔒 Very important! Leaving files open can cause:
  - Memory leaks
  - File lock issues
  - Data not saving properly (in write mode)

FILE READING

FILE HANDLING

```python
file = open('greetings.txt', 'r')

print(file.read())         # All at once
file.seek(0)               # Go back to start
print(file.readline())     # Just the first line
file.seek(0)
print(file.readlines())    # List of all lines

file.close()
```

```
Hello
How are you?
Goodbye
```

Method	Description	Sample Output
file.read()	Reads the **entire file** as one string	'Hello\nHow are you?\nGoodbye'
file.readline()	Reads the next **line** only	'Hello\n'
file.readlines()	Reads all lines into a **list**	['Hello\n', 'How are you?\n', 'Goodbye']

FILE WRITING

FILE HANDLING

```python
file = open('output.txt', 'w')

file.write("Line 1\n")                # Writes a single line
file.write("Line 2\n")                # Another write call
file.writelines(["Line 3\n", "Line 4\n"])  # Writes multiple lines from a list

file.close()
```

file.write(string)

- It writes exactly one string to the file.

file.writelines(list_of_strings)

- It writes a list of strings to the file
- Does not add newlines automatically — you must include **\n** in each string

Mode	**Description**
'w'	Write (overwrite existing file)
'a'	Append (add to end of file)

FILE ERROR HANDLING

FILE HANDLING

```python
try:
    file = open('message.txt', 'r')      # Try to open the file
    content = file.read()                # Try to read it
    print(content)
except FileNotFoundError:
    print("❌ File not found!")
except PermissionError:
    print("❌ No permission to read the file!")
finally:
    file.close()                         # Always close the file
```

Real files might not exist, or something might go wrong while reading.
We use try/except to prevent crashes.

SAFE FILE HANDLING

FILE HANDLING

```python
try:
    with open('message.txt', 'r') as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("❌ File not found!")
except PermissionError:
    print("❌ No permission to read the file!")
```

with open(...) as file:

- Auto-closes the file — you don’t need to call file.close()
- Prevents bugs if your code crashes while reading/writing
- Cleaner syntax - less boilerplate, more readable

QUICK RECAP - LIST COLLECTION

FILE HANDLING

What’s a List?

- A list is an ordered collection of items
- You can store anything: numbers, strings, even other lists!

```
fruits = ['apple', 'banana', 'cherry']
```

Key Properties:

- A list stores items in a specific order.
- You access items by their index number (e.g. my_list[0]).
- Lists are mutable, so you can change, add, or remove items.
- They can hold any data type, even mixed types.
- Useful when order matters or you need to loop over items.

QUICK RECAP - DICT COLLECTION

FILE HANDLING

What’s a Dictionary?

- A dictionary is an unordered collection of key-value pairs
- Think of it like a labeled shelf of folders — each label (key) tells you what’s inside (value)

```
person = {
    "name": "Alice",
    "age": 30,
    "email": "alice@example.com"
}
```

Key Properties:

- A dictionary stores key-value pairs, like labels and data.
- You access items using a key, not a position (e.g. person["name"]).
- Dictionaries are also mutable — you can update or delete entries.
- Keys are unique and values can be any data type.
- Great when you need to look up data quickly by name/label.

WHAT IS JSON?

FILE HANDLING

JSON stands for **JavaScript Object Notation**
It's a **text-based format** for storing and sharing **structured data.**

Why is it useful?

- Easy to read and write
- Works well with Python dicts and lists
- Common in APIs, configs, and data storage

JSON is just plain text!
You can save it in a .json file, and it will look like below:

```
{
  "name": "Alice",
  "age": 30,
  "skills": ["Python", "Data"]
}
```

PYTHON DATA COMPARE TO JSON

FILE HANDLING

Python Dict Structure

```python
person = {
    "name": "Alice",
    "age": 30,
    "is_student": False,
    "hobbies": ["reading", "biking"],
    "notes": None
}
```

Json Structure

```json
{
  "name": "Alice",
  "age": 30,
  "is_student": false,
  "hobbies": ["reading", "biking"],
  "notes": null
}
```

READING A JSON FILE

FILE HANDLING

What’s Happening:

- open('data.json', 'r') opens the file in read mode
- json.load(file) converts the JSON content into Python Dict
- Now you can access it like any Python data structure

```python
import json

with open('data.json', 'r') as file:
    data = json.load(file)

print(data)
```

WRITING JSON TO A FILE

FILE HANDLING

What’s Happening:

- json.dump(person, file) writes the Python object as JSON
- indent=2 makes the output nicely formatted (easy to read)
- The file output.json now contains JSON you can open anywhere

```python
import json

person = {
    "name": "Bob",
    "age": 25,
    "skills": ["JavaScript", "React"]
}

with open('output.json', 'w') as file:
    json.dump(person, file, indent=2)
```

OBJECT-ORIENTED PROGRAMMING

WHAT IS OOP?

OBJECT-ORIENTED

OOP (Object-Oriented Programming) is a programming approach where you model things in your code as objects — just like in the real world.

Why use OOP?

- Helps organize complex programs
- Makes your code reusable and easier to maintain
- Models real-world things clearly (like students, books, cars)

THINK IN OBJECTS

OBJECT-ORIENTED

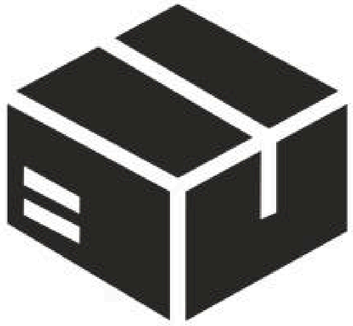
Imagine you're modeling a Student in code 💡 :
Real-world Student 🧍 :

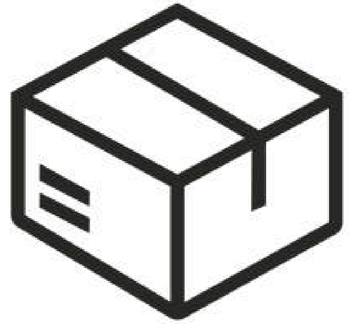
- Has a name, age, and list of courses
- Can enroll in a course
- Can show their info

```
Student
├── name          # e.g., "Alice"
├── age           # e.g., 21
└── courses       # e.g., ["Math", "Biology"]

Methods:
├── enroll(course)          # Adds a course
└── display_info()          # Prints details
```

Object Thinking = Grouping related data + behavior
"If something has data and can do things with it... it can probably be a class."





CLASS BASICS

OBJECT-ORIENTED

BLUEPRINT

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

What's Happening?

- **class Student:** → defines a new class
- **__init__** → runs when you create a new student (the constructor)
- **self.name, self.age** → store data (called attributes)
- **self** → refers to this specific student object

TEST CASE

```python
s1 = Student("Alice", 21)
print(s1.name)  # Alice
print(s1.age)   # 21
```

ADDING METHODS

OBJECT-ORIENTED

```python
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.courses = []

    def enroll(self, course):
        self.courses.append(course)

    def display_info(self):
        print(f"{self.name}, Age: {self.age}, Courses: {self.courses}")
```

TEST CASE

```python
s1 = Student("Bob", 22)
s1.enroll("Math")
s1.enroll("Physics")
s1.display_info()
```

OUTPUT

```
Bob, Age: 22, Courses: ['Math', 'Physics']
```

THINK IN OBJECTS — VEHICLE EXAMPLE

OBJECT-ORIENTED

🏗️ Class = Blueprint

- A Vehicle class defines what a vehicle is
- **Contains:**
  - **Attributes (state):** color, speed, fuel
  - **Methods (behavior):** move(), brake(), refuel()

🏭 Constructor = Factory

- **__init__()** is the factory that builds a new car
- It gives your object its **starting state** (like paint and fuel)

```python
class Vehicle:
    def __init__(self, color, fuel):
        self.color = color
        self.fuel = fuel
        self.speed = 0

    def move(self):
        self.speed += 10

    def brake(self):
        self.speed = 0
```

THINK IN OBJECTS — VEHICLE EXAMPLE

OBJECT-ORIENTED

```
car1 = Vehicle("red", 100)
car1.move()
```

Object = A Real Car 🚗

- **car1** is a real, specific vehicle (**an object**)
- It uses the **blueprint** from **Vehicle**

🧠 Object Thinking:
Blueprint (class) → Factory (__init__) → Real Thing (object)

THINK IN OBJECTS — INHERITANCE

OBJECT-ORIENTED

Base Class - Vehicle

```python
class Vehicle:
    def __init__(self, color, fuel):
        self.color = color
        self.fuel = fuel

    def drive(self):
        print("Driving...")
```

- Blueprint for all vehicles
- Has shared attributes like color, fuel
- Has basic behavior like drive()

Subclass - ElectricCar

```python
class ElectricCar(Vehicle):
    def __init__(self, color, battery_level):
        super().__init__(color, fuel=None)
        self.battery_level = battery_level

    def charge(self):
        print("Charging battery...")
```

- Inherits from Vehicle using **super()**
- Overrides fuel logic with battery_level
- Adds new behavior: charge()

THINK IN OBJECTS — POLYMORPHISM

OBJECT-ORIENTED

Polymorphism = Same method name, different behavior

```python
class Vehicle:
    def drive(self):
        print("The vehicle moves.")

class ElectricCar(Vehicle):
    def drive(self):
        print("The electric car glides silently.")
```

TEST CASE

```python
vehicles = [Vehicle(), ElectricCar()]

for v in vehicles:
    v.drive()
```

OUTPUT

```
The vehicle moves.
The electric car glides silently.
```