🔷 `try` and `except` Tutorial

In Python, errors that occur during runtime are called **exceptions**. If they are not handled, they will stop your program. To handle them gracefully, we use `try` and `except`.

1. Basic Structure

```
try:
    # Code that may cause an error
    result = 10 / 0
except:
    # Code to run if an error happens
    print("An error occurred")

#Here, dividing by zero causes an exception, and the program jumps to `except`.
```

```
An error occurred
```

2. Catching Specific Exceptions

Instead of catching all errors, you can catch **specific exception types**. This is a better practice because it avoids hiding other bugs.

```
try:
    num = int("abc")  # invalid conversion
except ValueError:
    print("You must enter a number")
```

```
You must enter a number
```

3. Multiple Except Blocks

You can handle different types of errors separately:

```
try:
    value = 10 / int(input("Enter a number: "))
except ZeroDivisionError:
    print("You cannot divide by zero!")
except ValueError:
    print("That was not a valid number.")
```

4. Using `else`

The `else` block runs **if no exception occurs**:

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Invalid input!")
else:
    print(f"You entered {num}")
```

```
You entered 10
```

5. Using `finally`

The `finally` block runs **no matter what happens** (useful for cleanup like closing files):

```
try:
    file = open("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing program...")   # always runs
```

```
File not found!
Closing program...
```

6. Raising Exceptions Manually

You can raise your own exceptions with `raise`:

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative!")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[6], line 3
      1 age = -5
      2 if age < 0:
----> 3     raise ValueError("Age cannot be negative!")

ValueError: Age cannot be negative!
```

7. Custom Exceptions

You can create your own exception classes:

```
class NegativeAgeError(Exception):
    pass

try:
    age = -2
    if age < 0:
        raise NegativeAgeError("Age must be positive!")
except NegativeAgeError as e:
    print(e)
```

```
Age must be positive!
```

🔷 Exercise 1: Division

Task: Write a program that asks the user for two numbers and divides the first by the second.

- Handle the case when the user enters a non-number.
- Handle division by zero.

🔷 Exercise 2: File Reading

Task: Write a program that tries to open a file called `data.txt` and prints its content.

- Handle the case when the file does not exist.
- Always print `"Program finished"` at the end.

🔷 Exercise 3: List Indexing

Task: You have a list: `numbers = [1, 2, 3, 4, 5]`

- Ask the user for an index and print the element at that index.
- Handle the case when the user enters a non-integer.
- Handle the case when the index is out of range.

🔷 Exercise 4: Multiple Exceptions

Task: Write a program that asks the user to enter a number and divides 100 by that number.

- Handle both `ValueError` and `ZeroDivisionError` separately.
- Print a success message if no error occurs.

🔷 Exercise 5: Raising Exceptions

Task: Write a program that asks the user to enter their age.

- If the age is negative, raise a `ValueError` with the message `"Age cannot be negative"`.
- Handle the exception and print the message.

🔷 Solutions

```
# Solution 1
try:
    a = float(input("Enter first number: "))
    b = float(input("Enter second number: "))
    print("Result:", a / b)
except ValueError:
    print("Please enter a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")

# Solution 2
try:
    file = open("data.txt", "r")
    print(file.read())
except FileNotFoundError:
    print("File not found!")
finally:
    print("Program finished")

# Solution 3
numbers = [1, 2, 3, 4, 5]
try:
    index = int(input("Enter index: "))
    print("Element:", numbers[index])
except ValueError:
    print("Invalid input, enter an integer!")
except IndexError:
    print("Index out of range!")

# Solution 4
try:
    num = int(input("Enter a number: "))
    result = 100 / num
except ValueError:
    print("You must enter a number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Success! Result is", result)

# Solution 5
try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise ValueError("Age cannot be negative")
except ValueError as e:
    print("Error:", e)
```

```
Start coding or generate with AI.
```

🔷 Python Functions

A **function** in Python is a block of reusable code that performs a specific task. Functions help make code modular, readable, and maintainable.

1. Defining a Function

The `def` keyword is used to define a function:

```
def greet():
    print("Hello, World!")
```

- `def` → defines a function
- `greet` → function name
- `()` → parameters (empty here)
- Code inside is indented

Calling the function:

```
greet()  # Output: Hello, World!
```

```
Hello, World!
```

2. Functions with Parameters

Functions can take **inputs** called parameters:

```
def greet_user(name):
    print(f"Hello, {name}!")

greet_user("Alice")  # Output: Hello, Alice!
greet_user("Bob")    # Output: Hello, Bob!
```

```
Hello, Alice!
Hello, Bob!
```

- `name` is a parameter
- Arguments are values we pass when calling the function

3. Return Values

Functions can **return values** using `return`:

```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)  # Output: 8
```

```
8
```

Without `return`, a function returns `None` by default.

4. Default Parameters

You can assign **default values** to parameters:

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

greet("Alice")  # Hello, Alice!
greet()         # Hello, Guest!
```

```
Hello, Alice!
Hello, Guest!
```

5. Keyword Arguments

You can pass arguments by **position** or by **name**:

```
def describe_pet(name, species="dog"):
    print(f"{name} is a {species}")
```

```
describe_pet("Buddy")             # Buddy is a dog
describe_pet(name="Kitty")        # Kitty is a dog
describe_pet(species="cat", name="Milo")  # Milo is a cat
```

```
Buddy is a dog
Kitty is a dog
Milo is a cat
```

6. Variable-Length Arguments

a) *args – Variable Positional Arguments

*args allows a function to accept any number of positional arguments.

Inside the function, args is treated as a tuple.

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))   # Output: 6
print(add_numbers(4, 5, 6, 7)) # Output: 22
```

```
6
22
```

```
def greet(*names):
    for name in names:
        print(f"Hello, {name}!")

greet("Alice", "Bob", "Charlie")
```

```
Hello, Alice!
Hello, Bob!
Hello, Charlie!
```

`*args` collects all extra positional arguments as a **tuple**

You can combine *args with normal parameters, but *args must come after regular parameters:

```
def greet(message, *names):
    for name in names:
        print(f"{message}, {name}!")

greet("Hi", "Alice", "Bob")
```

```
Hi, Alice!
Hi, Bob!
```

b) **kwargs → for multiple keyword arguments

- **kwargs allows a function to accept any number of keyword arguments.
- Inside the function, kwargs is treated as a **dictionary**.

```
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25)
```

```
name: Alice
age: 25
```

```
def user_profile(role, *args, **kwargs):
    print(f"Role: {role}")
    print("Positional args:", args)
    print("Keyword args:", kwargs)

user_profile("Admin", "Alice", "Bob", age=30, city="Cairo")
```

```
Role: Admin
Positional args: ('Alice', 'Bob')
```

```
Keyword args: {'age': 30, 'city': 'Cairo'}
```

Order matters:

```
def f(a, b, *args, **kwargs):
    ...
```

- Positional arguments → a, b
- Extra positional → *args
- Keyword arguments → **kwargs

```
def flexible_function(required, *args, **kwargs):
    print("Required:", required)
    print("Additional args:", args)
    print("Additional kwargs:", kwargs)

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

```
Required: Hello
Additional args: (1, 2, 3)
Additional kwargs: {'name': 'Alice', 'age': 25}
```

Using * and ** to Unpack Arguments

You can unpack lists/tuples into *args and dictionaries into **kwargs.

```
def add(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
print(add(*numbers))  # Output: 6

data = {'a': 10, 'b': 20, 'c': 30}
print(add(**data))    # Output: 60
```

```
6
60
```

7. Scope of Variables

Local vs Global Variables

- Variables inside a function are **local**
- Variables outside are **global**
- You can use `global` keyword to modify a global variable:

```
x = 10  # global

def foo():
    y = 5  # local
    print("Inside:", x, y)

foo()        # Inside: 10 5
print(y)     # Error! y is local to foo
```

```
Inside: 10 5
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[14], line 8
      5     print("Inside:", x, y)
      7 foo()        # Inside: 10 5
----> 8 print(y)     # Error! y is local to foo

NameError: name 'y' is not defined
```

```
count = 0

def increment():
    global count
    count += 1

increment()
print(count)  # 1
```

```
1
```

8. Lambda Functions (Anonymous Functions)

- Lambda functions are anonymous (no `def`)
- Best used for **short operations**

For **small one-line functions**, Python allows `lambda`:

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

```
25
```

```
# With two arguments
multiply = lambda a, b: a * b
print(multiply(3, 4))  # Output: 12
```

```
12
```

9. Nested Functions

- Useful for encapsulation and closures

Functions can be **defined inside other functions**:

```
def outer():
    def inner():
        print("Inside inner function")
    inner()

outer()
```

```
Inside inner function
```

10. Returning Functions (Closures)

Functions can **return other functions**:

```
def outer(msg):
    def inner():
        print(msg)
    return inner

f = outer("Hello Closure")
f()  # Output: Hello Closure
# * This is called a **closure**
```

```
Hello Closure
```

11. Decorators (Advanced)

1. Functions are First-Class Objects

- In Python, functions can be:
- Assigned to variables
- Passed as arguments
- Returned from other functions

```
def greet():
    return "Hello!"

say_hello = greet   # assign function
print(say_hello())  # Output: Hello!
```

```
Hello!
```

This property allows decorators to exist.

2. Basic Decorator

A simple decorator wraps a function:

```
def decorator(func):
    def wrapper():
        print("Before the function runs")
        func()
        print("After the function runs")
    return wrapper

def say_hello():
    print("Hello!")

# Apply decorator manually
say_hello = decorator(say_hello)
say_hello()
```

```
Before the function runs
Hello!
After the function runs
```

3. Using the @ Syntax

Python provides a shorthand `@decorator`:

```
@decorator
def say_hello():
    print("Hello!")

say_hello()
# This is equivalent to: say_hello = decorator(say_hello)
```

```
Before the function runs
Hello!
After the function runs
```

4. Decorators with Arguments

To decorate functions that accept arguments, the wrapper function must also accept `*args` and `**kwargs`:

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function")
        result = func(*args, **kwargs)
        print("After function")
        return result
    return wrapper

@decorator
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

```
Before function
Hello, Alice!
After function
```

5. Returning Values

Decorators can also modify or return values:

```
def uppercase(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

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

print(greet("Alice"))  # Output: HELLO, ALICE
```

```
HELLO, ALICE
```

6. Stacking Decorators

- Decorators are applied bottom to top.
- You can apply multiple decorators to a single function:

```
def bold(func):
    def wrapper(*args, **kwargs):
        return f"<b>{func(*args, **kwargs)}</b>"
    return wrapper

def italic(func):
    def wrapper(*args, **kwargs):
        return f"<i>{func(*args, **kwargs)}</i>"
    return wrapper

@bold
@italic
def greet(name):
    return f"Hello, {name}"

print(greet("Alice"))  # Output: <b><i>Hello, Alice</i></b>
```

```
<b><i>Hello, Alice</i></b>
```

7. Decorators with Arguments (Parameterized Decorators)

Sometimes you want the decorator itself to accept arguments:

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                func(*args, **kwargs)
        return wrapper
    return decorator
```

```
@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

```
Hello, Alice!
Hello, Alice!
Hello, Alice!
```

8. Preserving Metadata with functools.wraps

Without functools.wraps, the decorated function loses its name, docstring, and metadata:

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Before function")
        return func(*args, **kwargs)
    return wrapper

@decorator
def greet():
    """This function says hello"""
    print("Hello!")

print(greet.__name__)  # Output: greet
print(greet.__doc__)   # Output: This function says hello
```

9. Practical Examples

a) Logging

```
def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with args={args}, kwargs={kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log
def add(a, b):
    return a + b

print(add(2, 3))
```

```
import time

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timer
def compute():
    time.sleep(1)
    print("Done computing!")

compute()
```

12. Recursion

A function can **call itself**:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))  # Output: 120
```

```
120
```

13. Type Hints (Python 3.5+)

You can **hint the type** of parameters and return value:

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

print(greet("Alice"))
```

14. Docstrings (Documentation Strings)

Document your functions using `"""`:

```
def add(a, b):
    """
    This function returns the sum of a and b.
    Parameters:
        a (int or float)
        b (int or float)
    Returns:
        int or float
    """
    return a + b

help(add)
```

```
Help on function add in module __main__:

add(a, b)
    This function returns the sum of a and b.
    Parameters:
        a (int or float)
        b (int or float)
    Returns:
        int or float
```

✅ Summary Table

Concept	Example	Notes
Define Function	`def foo():`	Basic structure
Parameters	`def greet(name):`	Inputs to function
Return	`return a+b`	Sends value out
Default Arguments	`def f(x=5)`	Optional parameters
*args	`def f(*args)`	Variable positional args
**kwargs	`def f(**kwargs)`	Variable keyword args
Global & Local	`global var`	Scope rules
Lambda	`lambda x: x+1`	Anonymous function
Nested Functions	`def outer(): def inner():`	Encapsulation
Closure	`return inner`	Function returning function
Decorator	`@decorator`	Modify function behavior
Recursion	`def fact(n): ... fact(n-1)`	Function calls itself
Type Hints	`def f(x: int) -> int`	Optional typing
Docstrings	`"""docstring"""`	Documentation

```
Start coding or generate with AI.
```

🔷 Comprehensive Python Questions

Question 1: Student Grades Analyzer

Scenario: You are asked to create a program to analyze student grades for a class.

Requirements:

- Define a **dictionary** where the keys are student names and values are lists of grades (integers between 0–100).
- Write a function `average_grade(grades)` that calculates the average grade for a student.
- Loop through all students and print their **name, grades, and average**.
- Create a **set** of students who scored above 90 in any subject.
- Save the **summary report** to a text file `grades_report.txt`.

Skills Covered: Variables, dictionary, list, function, loop, set, file operations, conditionals.

Question 2: Text File Word Processor

Scenario: You are given a text file `input.txt` containing several lines of text.

Requirements:

- Read the file line by line.
- Create a function `word_count(text)` that returns the number of words in a line.
- For each line, print: `"Line <n>: <number_of_words> words"`.
- Find all **unique words** in the file (ignore case) and store them in a **set**.
- Save these unique words in a file `unique_words.txt`, one word per line.

Skills Covered: File operations, function, string operations, loops, set, case handling.

Question 3: Inventory Management System

Scenario: You are designing a simple inventory system for a store.

Requirements:

- Represent the inventory as a **list of dictionaries**, each containing `name`, `quantity`, and `price`.
- Write functions:
  - `total_value(item)` → returns total value (`quantity * price`) of one item.
  - `inventory_value(inventory)` → returns total value of all items.
- Print a table showing each item, quantity, price, and total value.
- Find the **most expensive item** and **item with highest quantity** using tuple unpacking and `max()` function.

Skills Covered: List, dictionary, tuple unpacking, functions, loops, conditionals, built-in functions.

Question 4: Number Analysis Tool

Scenario: Create a program that analyzes a list of numbers input by the user.

Requirements:

- Ask the user to input a series of numbers separated by commas and convert them into a **list of integers**.
- Write functions:
  - `even_numbers(numbers)` → returns a list of even numbers.
  - `odd_numbers(numbers)` → returns a list of odd numbers.
  - `prime_numbers(numbers)` → returns a list of prime numbers.
- Calculate and print: **sum, max, min, average** of the numbers.
- Store the **results in a dictionary** with keys `'sum'`, `'max'`, `'min'`, `'average'`, `'even'`, `'odd'`, `'prime'`.

Skills Covered: Variables, input handling, lists, loops, conditionals, functions, dictionary, mathematical operations.

Perfect! We can replace **Question 5** with a new one that specifically covers **decorators** and `*args` / `kwargs`**, while keeping it detailed and realistic. Here's the updated version:

Question 5: Flexible Task Logger

Scenario: You are building a task logging system where any function performing a task should automatically log its execution details.

Requirements:

- Write a **decorator** `task_logger` that:
  - Prints `"Starting task: <function_name>"` before the function runs.
  - Prints `"Finished task: <function_name>"` after the function runs.
  - Prints `"Arguments: <args>"` and `"Keyword Arguments: <kwargs>"`.
- Create at least **three different functions** representing tasks:
  1. `send_email(recipient, subject)` → prints `"Email sent to <recipient> with subject <subject>"`.
  2. `generate_report(name, pages=1)` → prints `"Report <name> generated with <pages> pages"`.
  3. `backup_files(*files)` → prints `"Backing up files: <file list>"`.
- Apply the `task_logger` decorator to all three functions.
- Call the functions with **different combinations of positional and keyword arguments** to demonstrate that the decorator correctly logs everything.

Skills Covered:

- Function decorators
- `*args` and `**kwargs`
- Function arguments (positional and keyword)
- Printing and string formatting
- Function metadata (`__name__`)

```
Start coding or generate with AI.
```

Question 1: Student Grades Analyzer

```
# Define the student grades dictionary
students = {
    "Alice": [85, 92, 78],
    "Bob": [95, 88, 91],
    "Charlie": [70, 65, 80],
    "Diana": [100, 98, 92]
}

# Function to calculate average grade
def average_grade(grades):
    return sum(grades) / len(grades)

# Set to store students who scored above 90 in any subject
high_achievers = set()

# Open a file to save the summary report
with open("grades_report.txt", "w") as file:
    for student, grades in students.items():
        avg = average_grade(grades)
        print(f"{student}: Grades = {grades}, Average = {avg:.2f}")
        file.write(f"{student}: Grades = {grades}, Average = {avg:.2f}\n")

        if any(grade > 90 for grade in grades):
            high_achievers.add(student)

print("Students who scored above 90 in any subject:", high_achievers)
```

Question 2: Text File Word Processor

```
# Function to count words in a line
def word_count(text):
    words = text.split()
    return len(words)

unique_words = set()

# Read the input file line by line
with open("input.txt", "r") as infile, open("unique_words.txt", "w") as outfile:
    for i, line in enumerate(infile, start=1):
        count = word_count(line)
        print(f"Line {i}: {count} words")

        # Add lowercase words to the set for uniqueness
        for word in line.strip().split():
            unique_words.add(word.lower())

    # Save unique words to file
    for word in sorted(unique_words):
        outfile.write(word + "\n")

print("Unique words saved to unique_words.txt")
```

Question 3: Inventory Management System

```
# Inventory represented as list of dictionaries
inventory = [
    {"name": "Laptop", "quantity": 5, "price": 800},
    {"name": "Mouse", "quantity": 50, "price": 20},
    {"name": "Keyboard", "quantity": 30, "price": 35},
    {"name": "Monitor", "quantity": 10, "price": 150}
]

# Function to calculate total value of one item
def total_value(item):
    return item['quantity'] * item['price']

# Function to calculate total inventory value
def inventory_value(inventory):
    return sum(total_value(item) for item in inventory)

print("Inventory Table:")
print(f"{'Item':10} {'Qty':>5} {'Price':>7} {'Total':>7}")
for item in inventory:
    total = total_value(item)
    print(f"{item['name']:10} {item['quantity']:5} {item['price']:7} {total:7}")

# Find most expensive item
most_expensive = max(inventory, key=lambda x: x['price'])
print("Most expensive item:", most_expensive['name'])

# Find item with highest quantity
highest_qty = max(inventory, key=lambda x: x['quantity'])
print("Item with highest quantity:", highest_qty['name'])

print("Total inventory value:", inventory_value(inventory))
```

Question 4: Number Analysis Tool

```
# Input numbers from user
numbers_input = input("Enter numbers separated by commas: ")
numbers = [int(num.strip()) for num in numbers_input.split(",")]
```

```
# Function to get even numbers
def even_numbers(nums):
    return [n for n in nums if n % 2 == 0]

# Function to get odd numbers
def odd_numbers(nums):
    return [n for n in nums if n % 2 != 0]

# Function to get prime numbers
def prime_numbers(nums):
    primes = []
    for n in nums:
        if n > 1 and all(n % i != 0 for i in range(2, int(n**0.5)+1)):
            primes.append(n)
    return primes

# Calculations
results = {
    "sum": sum(numbers),
    "max": max(numbers),
    "min": min(numbers),
    "average": sum(numbers)/len(numbers),
    "even": even_numbers(numbers),
    "odd": odd_numbers(numbers),
    "prime": prime_numbers(numbers)
}

# Print results
for key, value in results.items():
    print(f"{key.capitalize()}: {value}")
```

Question 5: Flexible Task Logger (Decorator + `*args`, `kwargs`)**

```
import functools

# Decorator to log task execution
def task_logger(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"\nStarting task: {func.__name__}")
        print(f"Arguments: {args}")
        print(f"Keyword Arguments: {kwargs}")
        result = func(*args, **kwargs)
        print(f"Finished task: {func.__name__}")
        return result
    return wrapper

# Task functions
@task_logger
def send_email(recipient, subject):
    print(f"Email sent to {recipient} with subject '{subject}'")

@task_logger
def generate_report(name, pages=1):
    print(f"Report '{name}' generated with {pages} pages")

@task_logger
def backup_files(*files):
    print(f"Backing up files: {files}")

# Test the functions
send_email("alice@example.com", subject="Meeting Reminder")
```