Python Raising Exceptions

So far, we have learned how Python **handles errors** using `try` and `except`.

But sometimes we don't want to wait for Python to discover a problem automatically.

Sometimes **we want to tell Python that something is wrong ourselves**.

That is where the `raise` statement comes in.

---

1. What Does "Raise an Exception" Mean?

Raising an exception means:

> **Manually telling Python that an error condition has occurred.**

We use the `raise` keyword.

```
raise ValueError("Invalid value")
```

Python immediately creates the specified exception and stops the normal flow of the current code unless that exception is handled.

---

2. Why Would We Raise an Exception?

Python automatically raises exceptions for many problems.

For example:

```
number = 10 / 0
```

Python automatically raises:

```
ZeroDivisionError
```

But many problems are **rules defined by our application**, not Python.

For example:

```
Age cannot be negative.
Password must be at least 8 characters.
Price cannot be below zero.
Username cannot be empty.
A user cannot withdraw more money than their balance.
```

Python doesn't automatically know these rules.

We can enforce them ourselves using `raise`.

---

3. Basic Syntax

The basic syntax is:

```
raise ExceptionType("Error message")
```

Example:

```
raise ValueError("Age cannot be negative")
```

This tells Python:

> Raise a `ValueError` with this message.

---

4. Simple Example

```
age = -5

if age < 0:
    raise ValueError("Age cannot be negative")

print("Valid age")
```

Output:

```
ValueError: Age cannot be negative
```

The `print()` statement never runs because the exception stops normal execution.

---

5. `raise` With User Input

Consider:

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

if age < 0:
```

```
    raise ValueError("Age cannot be negative")

print(f"Your age is {age}")
```

If the user enters:

```
20
```

Output:

```
Your age is 20
```

If the user enters:

```
-5
```

Python raises:

```
ValueError: Age cannot be negative
```

---

6. `raise` vs Automatic Exceptions

There are two ways an exception can happen.

Python raises it automatically

```
10 / 0
```

Python detects the invalid operation and raises:

```
ZeroDivisionError
```

We raise it manually

```
age = -5

if age < 0:
    raise ValueError("Age cannot be negative")
```

We detected the invalid condition and told Python to raise an exception.

---

7. Raising `ValueError`

`ValueError` is useful when the **type of data is correct but the value is not acceptable**.

Example:

```
age = 150

if age > 120:
    raise ValueError("Age is not realistic")
```

The value is an integer, so its type is fine.

The problem is the value itself.

Another example:

```
percentage = 150

if percentage > 100:
    raise ValueError("Percentage cannot be greater than 100")
```

---

8. Raising `TypeError`

`TypeError` is useful when an operation receives an inappropriate type.

Example:

```
def calculate_total(price):
    if not isinstance(price, (int, float)):
        raise TypeError("Price must be a number")

    return price * 2
```

Now:

```
calculate_total("100")
```

raises:

```
TypeError: Price must be a number
```

---

9. Raising `TypeError` vs `ValueError`

This distinction is important.

Wrong type

```
calculate_total("100")
```

Use:

raise TypeError(...)

Correct type, unacceptable value

calculate_total(-100)

Use:

raise ValueError(...)

Think:

TypeError Wrong kind of data
ValueError Right kind of data, wrong value

---

10. Raising ZeroDivisionError

You can also manually raise other built-in exceptions.

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")

    return a / b
```

Then:

print(divide(10, 0))

raises:

ZeroDivisionError: Cannot divide by zero

Python would naturally raise the same type if the division itself happened, but manually checking first can allow you to provide clearer application-specific messages.

---

11. Raising an Exception Inside a Function

Functions are one of the most common places to use `raise`.

```
def withdraw(balance, amount):
    if amount < 0:
        raise ValueError("Amount cannot be negative")

    if amount > balance:
        raise ValueError("Insufficient balance")

    return balance - amount
```

Now:

```
balance = withdraw(1000, 300)

print(balance)
```

Output:

```
700
```

But:

```
withdraw(1000, 1500)
```

raises an exception.

---

12. raise and try / except

We can raise an exception and then handle it.

```
try:
    age = -10

    if age < 0:
        raise ValueError("Age cannot be negative")

except ValueError as error:
    print(error)
```

Output:

```
Age cannot be negative
```

The flow is:

```
Check condition
      
Invalid?
      
raise ValueError
      
except catches it
      
Handle the error
```

---

13. Why Not Just Use print()?

A beginner might write:

```
age = -10

if age < 0:
    print("Invalid age")
```

But the program continues.

```
age = -10

if age < 0:
    print("Invalid age")

print("Creating user...")
```

Output:

```
Invalid age
Creating user...
```

If an invalid age should prevent the operation, raising an exception is more appropriate:

```
age = -10

if age < 0:
    raise ValueError("Invalid age")

print("Creating user...")
```

Now the invalid state cannot silently continue through the program.

---

14. Validation With `raise`

A very common pattern is:

```
def create_user(username):
    if not username:
        raise ValueError("Username cannot be empty")

    return username
```

The function protects its own rules.

This is better than assuming every caller will validate the input correctly.

---

15. Multiple Validation Rules

You can raise different exceptions for different problems.

```
def register_user(username, age):
    if not username:
        raise ValueError("Username cannot be empty")

    if not isinstance(age, int):
        raise TypeError("Age must be an integer")

    if age < 13:
        raise ValueError("User must be at least 13 years old")

    return "User registered"
```

Now the function clearly communicates what went wrong.

---

16. `raise` Stops Normal Execution

Consider:

```
print("Start")

raise ValueError("Something went wrong")

print("End")
```

Output:

```
Start
ValueError: Something went wrong
```

`"End"` is never printed.

Once an exception is raised, Python looks for a matching exception handler.

---

17. Raising From Inside a Condition

This is one of the most common patterns:

```
if condition_is_invalid:
    raise ValueError("Invalid data")
```

For example:

```
password = "abc"
```

```
if len(password) < 8:
    raise ValueError("Password must contain at least 8 characters")
```

---

18. Raising From Inside a Loop

You can raise an exception inside a loop too.

```
numbers = [10, 20, -5, 30]

for number in numbers:
    if number < 0:
        raise ValueError("Negative numbers are not allowed")

    print(number)
```

Output:

```
10
20
ValueError: Negative numbers are not allowed
```

The loop stops when the exception is raised.

---

19. Raising From Inside a Class

Exceptions can also enforce rules inside classes.

```
class BankAccount:
    def __init__(self, balance):
        if balance < 0:
```

```
            raise ValueError("Balance cannot be negative")

        self.balance = balance
```

This prevents invalid objects from being created.

```
account = BankAccount(1000)
```

Works.

But:

```
account = BankAccount(-500)
```

raises:

```
ValueError: Balance cannot be negative
```

---

20. Raising Exceptions in a Real Application

Imagine haas.dev has a function that finds a resource.

```
def get_resource(resources, resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    raise ValueError("Resource not found")
```

Now the caller knows that the requested resource doesn't exist.

Instead of silently returning something unexpected, the function clearly communicates the problem.

A caller could handle it:

```
try:
    resource = get_resource(resources, "python-999")
except ValueError as error:
    print(error)
```

---

21. `raise` Without an Exception Object

You can also write:

```
raise ValueError
```

This creates a `ValueError` without a custom message.

But in application code, a useful message is usually better:

```
raise ValueError("Username cannot be empty")
```

A clear message makes debugging much easier.

---

22. Raising With Variables

Exception messages can contain useful information.

```
age = -5
```

```
if age < 0:
    raise ValueError(f"Invalid age: {age}")
```

Output:

```
ValueError: Invalid age: -5
```

This can be useful when debugging.

---

23. Raising Different Exceptions

A function can raise different exception types depending on the problem.

```
def process_age(age):
    if not isinstance(age, int):
        raise TypeError("Age must be an integer")

    if age < 0:
        raise ValueError("Age cannot be negative")

    return age
```

Now:

```
process_age("20")
```

raises `TypeError`.

While:

```
process_age(-20)
```

raises `ValueError`.

---

24. Catching Different Raised Exceptions

The caller can handle each exception separately.

```
try:
    age = process_age("20")

except TypeError:
    print("Wrong data type")

except ValueError:
    print("Invalid age")
```

This creates a clean separation between:

```
What went wrong?
        ↓
Which exception?
        ↓
How should the caller handle it?
```

---

25. Raising an Exception From Another Exception

Sometimes one exception happens because of another.

Python allows us to explicitly connect them using:

`raise ... from ...`

Example:

```
try:
    number = int("hello")
except ValueError as error:
    raise RuntimeError("Could not process user input") from error
```

This tells Python:

> The new exception was caused by the original exception.

This is called **exception chaining**.

It becomes especially useful in larger applications where low-level errors need to be converted into meaningful higher-level errors.

---

26. raise ... from None

Sometimes you don't want the original exception to be displayed as part of the exception chain.

You can use:

```
raise ValueError("Invalid input") from None
```

Example:

```
try:
    number = int("hello")
```

```
except ValueError:
    raise ValueError("Please enter a valid number") from None
```

The caller sees the cleaner application-level error instead of the original conversion error details.

---

27. Re-Raising an Exception

Inside an `except` block, you can use:

```
raise
```

This means:

> Raise the current exception again.

Example:

```
try:
    number = int("hello")

except ValueError:
    print("Logging the error...")
    raise
```

The program prints:

```
Logging the error...
```

and then the original `ValueError` continues upward.

This is called **re-raising an exception**.

---

28. Why Re-Raise Exceptions?

Imagine a lower-level function catches an error because it needs to log it.

```
def process_data():
    try:
        value = int("hello")
    except ValueError:
        print("Error logged")
        raise
```

The function records the problem but doesn't pretend the operation succeeded.

The caller can still handle the exception.

```
try:
    process_data()
except ValueError:
    print("Could not process data")
```

This is useful in larger programs.

---

29. `raise` vs `return`

These are completely different.

`return`

```
def check_age(age):
    if age < 0:
        return None
```

The function returns normally.

raise

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
```

The function signals that an exceptional condition occurred.

Think:

```
return  Normal result
raise   Exceptional condition
```

---

30. raise vs print

Again, these have different purposes.

```
print("Invalid password")
```

Only displays a message.

While:

```
raise ValueError("Invalid password")
```

creates an exception and changes the program's control flow.

---

31. When Should You Raise an Exception?

Use `raise` when:

1. A function receives invalid input

```
if amount < 0:
    raise ValueError("Amount cannot be negative")
```

2. A required condition is not satisfied

```
if not username:
    raise ValueError("Username required")
```

3. An operation cannot continue safely

```
if balance < amount:
    raise ValueError("Insufficient balance")
```

4. A program reaches an invalid state

```
if user is None:
    raise ValueError("User is required")
```

---

32. When Should You Not Raise an Exception?

Don't use exceptions for ordinary program flow when a normal return value is clearer.

For example, if searching for an item is expected to fail frequently, you might return `None`:

```
def find_user(users, username):
    for user in users:
        if user["name"] == username:
            return user

    return None
```

Whether to return `None` or raise an exception depends on what the absence means in your application's design.

A useful question is:

> Is this a normal expected outcome, or is it an invalid/exceptional condition?

---

33. Common Mistake: Raising Too Much

Avoid code like:

```
def add(a, b):
    if a is None:
        raise ValueError("a missing")

    if b is None:
        raise ValueError("b missing")

    return a + b
```

if the function's contract already makes these conditions impossible or unnecessary to validate.

Exceptions should communicate meaningful violations of a function's contract, not every tiny condition.

---

34. Common Mistake: Using the Wrong Exception Type

Bad:

```
age = "twenty"

if not isinstance(age, int):
    raise ValueError("Age must be an integer")
```

Better:

```
if not isinstance(age, int):
    raise TypeError("Age must be an integer")
```

Because the problem is the **type**.

---

35. Common Mistake: Vague Error Messages

Avoid:

```
raise ValueError("Error")
```

Better:

```
raise ValueError("Password must contain at least 8 characters")
```

The second message tells the developer or user what went wrong.

---

36. Common Mistake: Catching Immediately Without Purpose

This:

```
try:
    raise ValueError("Invalid")
except ValueError:
    pass
```

hides the problem.

If an exception is caught, there should usually be a reason:

- recover from it
- show a useful message
- log it
- convert it
- retry
- clean up and re-raise

---

37. Validation Function Example

A clean validation function:

```
def validate_username(username):
    if not isinstance(username, str):
        raise TypeError("Username must be a string")

    if not username.strip():
        raise ValueError("Username cannot be empty")
```

```
    if len(username) < 3:
        raise ValueError("Username must contain at least 3 characters")

    return True
```

Now:

```
validate_username("Hafsa")
```

returns:

```
True
```

But:

```
validate_username("")
```

raises:

```
ValueError: Username cannot be empty
```

---

38. A Complete Application Example

Consider an order system:

```
def create_order(product, quantity):
    if not isinstance(quantity, int):
        raise TypeError("Quantity must be an integer")

    if quantity <= 0:
        raise ValueError("Quantity must be greater than zero")
```

```
    if not product:
        raise ValueError("Product is required")

    return {
        "product": product,
        "quantity": quantity
    }
```

The caller can handle the errors:

```
try:
    order = create_order("Python Course", 2)

except TypeError as error:
    print(f"Type error: {error}")

except ValueError as error:
    print(f"Validation error: {error}")

else:
    print("Order created:")
    print(order)
```

This is a realistic pattern used in application development.

---

39. `raise` and Program Architecture

In larger applications, different layers often have different responsibilities.

For example:

```
User Input
    ↓
Validation
    ↓
Business Logic
    ↓
Database
    ↓
Response
```

A lower-level function can raise an exception:

```
raise ValueError("Invalid resource ID")
```

A higher-level layer can decide how to respond:

```
try:
    resource = get_resource(resource_id)
except ValueError:
    return {"error": "Resource not found"}
```

This separation keeps individual functions focused.

---

40. Raising Exceptions in APIs

Imagine a backend API receives:

```
{
    "age": -5
}
```

The validation layer can detect the invalid value:

```
if age < 0:
    raise ValueError("Age cannot be negative")
```

The API layer can then convert that exception into an appropriate response.

This demonstrates an important software engineering principle:

> The function that detects the problem does not always have to decide how the problem is presented to the user.

---

41. Mini Project: User Registration Validator

Create a function that validates:

- username must be a string
- username cannot be empty
- username must contain at least 3 characters
- age must be an integer
- age must be at least 13

```
def register_user(username, age):

    if not isinstance(username, str):
        raise TypeError("Username must be a string")

    if not username.strip():
        raise ValueError("Username cannot be empty")

    if len(username) < 3:
        raise ValueError("Username must contain at least 3 characters")
```

```
    if not isinstance(age, int):
        raise TypeError("Age must be an integer")

    if age < 13:
        raise ValueError("User must be at least 13 years old")

    return "Registration successful"
```

Use it:

```
try:
    result = register_user("Hafsa", 25)

except TypeError as error:
    print(error)

except ValueError as error:
    print(error)

else:
    print(result)
```

---

42. 5 Minute Challenge

Create a function:

```
def withdraw(balance, amount):
    ...
```

Requirements:

- `balance` must be a number
- `amount` must be a number
- amount cannot be negative
- amount cannot be greater than balance
- raise appropriate exceptions
- return the remaining balance when everything is valid

Example:

```
withdraw(1000, 300)
```

Expected:

```
700
```

But:

```
withdraw(1000, 1500)
```

should raise an exception.

---

43. Exercises

Exercise 1

Write a function that raises `ValueError` if a number is negative.

Exercise 2

Write a function that raises `TypeError` if the input isn't a string.

Exercise 3

Create a function that validates a password.

Rules:

- must be a string
- minimum 8 characters
- must not be empty

Raise meaningful exceptions when the rules are violated.

Exercise 4

Create a function:

`def calculate_percentage(marks, total):`

Raise exceptions when:

- marks isn't a number
- total isn't a number
- total is zero
- marks is negative
- marks is greater than total

Return the percentage when valid.

Exercise 5

Write a program that raises an exception when a resource ID is not found.

---

44. Mini Quiz

1. Which keyword manually creates an exception?

A. `error`
B. `throw`
C. `raise`
D. `exception`

Answer: C

---

2. Which exception is generally appropriate for the wrong type?

A. `TypeError`
B. `ValueError`
C. `KeyError`
D. `IndexError`

Answer: A

---

3. Which exception is generally appropriate when the type is correct but the value is invalid?

A. `TypeError`
B. `ValueError`
C. `NameError`
D. `SyntaxError`

Answer: B

---

4. What happens after `raise`?

A. Python ignores it
B. Normal execution continues
C. Python raises an exception and searches for a handler
D. The program always restarts

Answer: C

---

5. What does `raise` inside an `except` block do?

```
except ValueError:
    raise
```

A. Deletes the exception
B. Re-raises the current exception
C. Converts it to `ValueError`
D. Stops exception handling permanently

Answer: B

---

45. Interview Questions

Beginner

1. What is the `raise` keyword in Python?
2. Why would you manually raise an exception?
3. How do you raise a `ValueError`?
4. What is the difference between `raise` and `print()`?
5. What happens after an exception is raised?

Intermediate

6. When should you use `ValueError`?
7. When should you use `TypeError`?
8. What is exception chaining?
9. What does `raise ... from ...` do?
10. What does a bare `raise` inside an `except` block do?

Practical

11. How would you validate function arguments using `raise`?
12. How would you design exceptions in a multi-layer application?
13. When should a function return `None` instead of raising an exception?
14. Why are meaningful exception messages important?
15. Why should you avoid catching an exception immediately without handling or re-raising it?

---

46. Quick Cheat Sheet

Basic raise

```
raise ValueError("Invalid value")
```

Type validation

```
if not isinstance(value, int):
    raise TypeError("Value must be an integer")
```

Value validation

```
if value < 0:
    raise ValueError("Value cannot be negative")
```

Handle a raised exception

```
try:
    validate()
except ValueError as error:
    print(error)
```

Re-raise

```
except ValueError:
    raise
```

Exception chaining

```
except ValueError as error:
    raise RuntimeError("Processing failed") from error
```

---

47. Key Takeaways

- `raise` lets you manually trigger an exception.
- Python automatically raises exceptions for many runtime problems, but your application has its own rules that Python cannot know.
- Use `ValueError` when the value is inappropriate.

- Use `TypeError` when the data type is inappropriate.
- A raised exception stops normal execution until a matching handler is found.
- `raise` is commonly used for validation and enforcing function contracts.
- `raise` can be used inside functions, classes, loops, and application layers.
- `raise` can re-raise an existing exception.
- `raise ... from ...` can connect a new exception to its original cause.
- Meaningful exception messages make debugging easier.
- Exceptions should represent genuinely exceptional or invalid conditions, not ordinary expected outcomes.

The core pattern is:

```
def process(value):
    if invalid_type(value):
        raise TypeError("Wrong type")

    if invalid_value(value):
        raise ValueError("Wrong value")

    return "Success"
```

Once you understand `raise`, exception handling becomes more than just **reacting to Python errors**. You can actively design your programs to detect invalid states, enforce rules, and communicate problems clearly.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

haas.dev
https://dev-roast-app.vercel.app