Python `try` and `except`

Introduction

Programs do not always receive perfect input or behave exactly as expected.

A user might enter:

`hello`

when your program expects a number.

A file might not exist.

A calculation might attempt to divide by zero.

Instead of allowing the program to crash, Python provides `try` and `except` for handling these situations.

The basic idea is:

`try` → attempt something
`except` → handle the problem if it happens

---

1. What Is `try` and `except`?

`try` and `except` are used for **exception handling**.

Basic structure:

```
try:
```

```
    # code that might cause an exception
except:
    # code that handles the exception
```

For example:

```
try:
    number = int(input("Enter a number: "))
except:
    print("Invalid input")
```

If the user enters:

```
25
```

the conversion succeeds.

If the user enters:

```
hello
```

the exception is handled.

---

2. Why Do We Need `try` and `except`?

Without exception handling:

```
number = int(input("Enter a number: "))
print(number)
```

If the user enters:

`abc`

Python raises:

`ValueError`

and the program stops.

With exception handling:

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

Now the program can respond to the problem instead of crashing immediately.

---

3. How `try` Works

The code inside `try` is executed normally.

Example:

```
try:
    number = 10
    print(number)
except ValueError:
    print("Something went wrong")
```

Output:

```
10
```

Because no exception occurred, the `except` block does not run.

---

4. How `except` Works

If an exception occurs inside `try`, Python stops executing the remaining code inside that `try` block and searches for a matching `except`.

Example:

```
try:
    number = int("hello")
    print("This will not run")
except ValueError:
    print("Invalid number")
```

Output:

```
Invalid number
```

The flow is:

```
try
 
int("hello")
 
ValueError
 
except ValueError
 
print("Invalid number")
```

---

5. Catching a Specific Exception

It is better to specify which exception you expect.

Instead of:

```
try:
    number = int(input("Enter number: "))
except:
    print("Error")
```

prefer:

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

This makes your code more precise.

---

6. `ValueError` Example

Consider:

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Age must be a number.")
```

If the user enters:

```
25
```

the program succeeds.

If the user enters:

```
twenty five
```

Python raises `ValueError`, and the `except` block handles it.

---

7. `ZeroDivisionError` Example

Another common exception is `ZeroDivisionError`.

```
try:
    number = 100 / 0
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

Output:

```
You cannot divide by zero.
```

The program handles the problem instead of stopping with an unhandled exception.

---

8. Handling Multiple Exceptions

A program can encounter different types of problems.

Example:

```
try:
    number = int(input("Enter a number: "))
    result = 100 / number

except ValueError:
    print("Please enter a valid number.")

except ZeroDivisionError:
    print("Number cannot be zero.")
```

Now:

```
Input □ Problem □ Response

abc   □ ValueError       □ Invalid number
0     □ ZeroDivisionError □ Cannot divide by zero
10    □ No exception     □ Calculation succeeds
```

---

9. Multiple Exceptions With One `except`

If different exceptions need the same response, you can group them.

```
try:
    number = int(input("Enter a number: "))
    result = 100 / number

except (ValueError, ZeroDivisionError):
    print("Invalid input.")
```

Both exceptions are handled by the same block.

---

10. The `as` Keyword

You can store the exception object in a variable.

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

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

The variable `error` contains information about the exception.

You can also write:

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

except ValueError as error:
    print("Something went wrong:", error)
```

---

11. What Happens After `except`?

After the matching `except` block finishes, Python continues with the code after the entire `try` and `except` structure.

Example:

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

except ValueError:
```

```
    print("Invalid number")

print("Program continues")
```

Output:

```
Invalid number
Program continues
```

This is one of the main benefits of exception handling.

---

12. Code After the Error Inside `try` Does Not Run

Consider:

```
try:
    number = int("hello")
    print("Step 2")
    print("Step 3")

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

Python never reaches:

```
print("Step 2")
```

because the exception happens first.

The flow is:

int("hello")

ValueError

skip remaining try code

except runs

---

13. The else Block

Although try and except are the main focus, Python also allows else.

else runs only when no exception occurs.

```
try:
    number = int(input("Enter a number: "))

except ValueError:
    print("Invalid input.")

else:
    print("You entered:", number)
```

If the user enters:

```
50
```

output:

```
You entered: 50
```

If the user enters:

```
abc
```

output:

```
Invalid input.
```

---

14. Why Use `else`?

The `else` block is useful when you want to separate:

```
Risky operation
    
try

Error handling
    
except

Successful operation
    
else
```

Example:

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

except ValueError:
    print("Invalid age.")

else:
    print("Age accepted.")
```

---

15. `try` Should Contain Focused Code

Avoid putting a huge amount of code inside `try`.

Instead of:

```
try:
    # hundreds of lines
    ...
except Exception:
    print("Error")
```

keep the risky operation focused:

```
try:
    age = int(user_input)

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

Then handle the rest separately.

This makes it easier to understand which operation caused the exception.

---

16. `try` and User Input

One of the most common uses of `try` and `except` is validating user input.

```
try:
```

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

A more useful version can repeatedly ask until the input is valid:

```
while True:
    try:
        age = int(input("Enter your age: "))
        break

    except ValueError:
        print("Please enter a valid number.")

print("Your age is:", age)
```

Now the program keeps asking until it receives a valid integer.

---

17. `try` and `except` With a Function

Exception handling can be used inside functions.

```
def get_number():
    try:
        return int(input("Enter a number: "))

    except ValueError:
        return None
```

Then:

```
number = get_number()
```

```
if number is None:
    print("Invalid input.")
else:
    print("Number:", number)
```

The function handles the conversion problem and returns a useful result.

---

18. Letting the Caller Handle the Exception

Sometimes it is better for a function to raise the exception instead of handling it.

Example:

```
def convert_number(value):
    return int(value)
```

Then another part of the program can decide what to do:

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

except ValueError:
    print("Could not convert the value.")
```

This gives the caller more control.

A useful design principle is:

> Handle an exception where you know what to do about it.

---

19. `try` and `except` With Files

Suppose a program needs to read a file.

```
try:
    with open("data.txt", "r") as file:
        data = file.read()

except FileNotFoundError:
    print("The file was not found.")
```

If the file exists, it is read normally.

If it does not exist, the exception is handled.

---

20. `try` and `except` With Dictionaries

Suppose:

```
user = {
    "name": "Hafsa",
    "age": 25
}
```

This can cause a `KeyError`:

```
try:
    print(user["email"])

except KeyError:
```

```
    print("Email is not available.")
```

Output:

```
Email is not available.
```

For simple dictionary lookups, `.get()` can often be cleaner:

```
print(user.get("email"))
```

So exception handling is not always the only solution.

---

21. `try` and `except` With Lists

Example:

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

try:
    print(numbers[10])

except IndexError:
    print("That index does not exist.")
```

The exception is:

```
IndexError
```

because index `10` is outside the list.

---

22. Catching `Exception`

Python allows:

```
try:
    risky_code()

except Exception as error:
    print("Error:", error)
```

This catches most ordinary application exceptions.

It can be useful when logging unexpected errors.

However, it should not automatically replace specific exception handling.

Prefer:

`except ValueError:`

when you know that `ValueError` is what you expect.

---

23. Why `except:` Is Usually Not Ideal

You may see:

```
try:
    risky_code()

except:
```

```
    print("Something went wrong")
```

This catches almost everything.

The problem is that it can hide problems you did not intend to handle.

Better:

```
try:
    number = int(user_input)

except ValueError:
    print("Invalid number.")
```

Now the code clearly communicates what problem it expects.

---

24. Avoid Silencing Exceptions

This is usually a bad pattern:

```
try:
    process_data()

except:
    pass
```

If `process_data()` fails, the program says nothing.

You may not know:

- what failed

- why it failed
- whether important data was lost
- whether the application is still working correctly

Instead:

```
try:
    process_data()

except ValueError as error:
    print("Invalid data:", error)
```

Or log the exception if the application requires proper error reporting.

---

25. Nested `try` and `except`

You can technically place one `try` inside another:

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

    except ValueError:
        print("Invalid conversion")

except Exception:
    print("Outer handler")
```

But nested exception handling should be used only when the different levels have meaningful responsibilities.

Do not nest `try` blocks simply because you can.

---

26. Exception Handling vs Validation

These concepts are related but different.

Validation

Checks whether data follows expected rules.

```
age = 20

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

Exception handling

Handles unexpected runtime problems.

```
try:
    age = int(user_input)

except ValueError:
    print("Age must be a number.")
```

A strong program often uses both.

---

27. Preventing an Exception vs Handling It

Sometimes you can prevent a problem:

```
if number != 0:
```

```
    result = 100 / number
```

Other times, handling the exception is appropriate:

```
try:
    result = 100 / number

except ZeroDivisionError:
    print("Cannot divide by zero.")
```

The right approach depends on the situation.

---

28. Real World Example: Login Input

Suppose a program expects a user ID.

```
try:
    user_id = int(input("Enter your user ID: "))

except ValueError:
    print("User ID must contain numbers.")

else:
    print("Searching for user:", user_id)
```

The code clearly separates:

```
Input conversion
      
try

Invalid input
```

```
    □
except

Successful conversion
    □
else
```

---

29. Real World Example: haas.dev Resource Search

Imagine a resource search function:

```
def search_resources(resources, query):
    if not isinstance(query, str):
        raise TypeError("Search query must be a string")

    return [
        resource
        for resource in resources
        if query.lower() in resource["title"].lower()
    ]
```

The calling code can safely handle invalid input:

```
try:
    results = search_resources(resources, 123)

except TypeError as error:
    print("Search error:", error)

else:
    print("Results:", results)
```

Here:

- the function detects invalid data
- `TypeError` describes the problem
- `except` decides how the application responds

This separation becomes valuable as applications grow.

---

30. `try` / `except` With Your Own Modules

Suppose:

`calculator.py`

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

    return a / b
```

`main.py`

```
from calculator import divide

try:
    result = divide(20, 0)

except ValueError as error:
    print("Calculation error:", error)

else:
    print("Result:", result)
```

The module defines the error condition.

The calling program decides how to handle it.

---

31. Common Mistakes

Mistake 1: Catching every exception

```
except:
    ...
```

This can hide unexpected problems.

Mistake 2: Empty exception handling

```
except ValueError:
    pass
```

The error disappears without explanation.

Mistake 3: Huge `try` blocks

```
try:
    # entire application
```

This makes debugging difficult.

Mistake 4: Wrong exception type

If code can raise:

`ValueError`

but you only catch:

`except KeyError:`

the exception remains unhandled.

Mistake 5: Using exceptions for normal conditions

Do not deliberately cause exceptions for simple decisions.

Prefer:

```
if user is None:
    print("User not found")
```

when that is a normal expected condition.

---

32. Best Practices

1. Catch specific exceptions

`except ValueError:`

is usually clearer than:

`except:`

2. Keep `try` blocks small

Put only the potentially failing operation inside.

3. Give useful error messages

`print("Please enter a valid number.")`

is more useful than:

`print("Error")`

4. Do not silently ignore failures

Avoid unnecessary:

`pass`

inside exception handlers.

5. Use `else` for successful execution

It keeps success logic separate from error handling.

6. Handle exceptions at the appropriate level

The code that understands how to respond should usually handle the exception.

---

33. Problem Solving Framework

Whenever you write code that might fail, ask:

What can go wrong?
↓
Which exception can occur?
↓
Can I prevent the problem?
↓
If not, how should I handle it?
↓
What should the user see?

For example:

User enters "abc"
↓
int() cannot convert it
↓
ValueError
↓
except ValueError
↓
"Please enter a valid number."

---

34. Mini Project: Safe Calculator

Create:

calculator.py

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

    return a / b
```

Create:

main.py

```
from calculator import divide

try:
    first = float(input("Enter first number: "))
    second = float(input("Enter second number: "))

    result = divide(first, second)

except ValueError as error:
    print("Error:", error)

else:
    print("Result:", result)
```

Test:

```
10
2
```

Output:

```
Result: 5.0
```

Test:

```
10
0
```

Output:

```
Error: Cannot divide by zero
```

---

35. Five Minute Challenge

Create a function:

```
def get_positive_number():
    ...
```

The function should:

1. Ask the user for a number.
2. Convert it to `int`.
3. Handle invalid input using `try` and `except`.
4. Reject numbers less than or equal to zero.
5. Keep asking until the user enters a valid positive number.

Expected behavior:

```
Enter a number: abc
Please enter a valid number.

Enter a number: -5
Number must be positive.

Enter a number: 10
Accepted: 10
```

---

36. Exercises

Exercise 1

Write a program that asks for two numbers and handles:

`ValueError`
`ZeroDivisionError`

Exercise 2

Create:

```
def get_list_item(items, index):
    ...
```

Use `try` and `except` to handle an invalid index.

Exercise 3

Create a dictionary:

```
student = {
    "name": "Hafsa",
    "course": "Python"
}
```

Try accessing `"email"` and handle the `KeyError`.

Exercise 4

Write a program that opens:

`notes.txt`

and handles `FileNotFoundError`.

Exercise 5

Create:

```
def convert_age(value):
    ...
```

Use `try` and `except` to safely convert the value into an integer.

---

37. Mini Quiz

Question 1

Which block contains code that might cause an exception?

A. `try`

B. `except`

C. `else`

D. `finally`

Answer: A

Question 2

Which block handles an exception?

A. `try`

B. `except`

C. `else`

D. `raise`

Answer: B

Question 3

What exception can occur here?

```
int("hello")
```

Answer: `ValueError`

Question 4

What exception can occur here?

`10 / 0`

Answer: `ZeroDivisionError`

Question 5

What does this do?

`except ValueError as error:`

It catches a `ValueError` and stores the exception object in `error`.

Question 6

When does `else` run?

Answer: When the `try` block completes without an exception.

---

38. Interview Questions

1. What is `try` used for?

It contains code that may raise an exception.

2. What is `except` used for?

It handles a matching exception raised inside the `try` block.

3. Can there be multiple `except` blocks?

Yes.

```
try:
    ...
except ValueError:
    ...
except TypeError:
    ...
```

4. Can one `except` handle multiple exceptions?

Yes.

```
except (ValueError, TypeError):
    ...
```

5. What does `as error` do?

It stores the exception object in a variable.

6. Why should specific exceptions usually be caught?

They make the program's error handling clearer and prevent unrelated problems from being hidden.

7. What happens to the remaining code inside `try` after an exception?

It is skipped, and Python moves to the matching `except` block.

8. Where does execution continue after `except`?

It continues with the code after the `try`/`except` structure.

---

39. Cheat Sheet

Basic

```
try:
    risky_code()

except ValueError:
    handle_error()
```

Multiple exceptions

```
try:
    risky_code()

except ValueError:
    handle_value_error()

except TypeError:
    handle_type_error()
```

Same handler

```
try:
    risky_code()

except (ValueError, TypeError):
    print("Invalid input")
```

Exception object

```
try:
    risky_code()

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

With `else`

```
try:
    risky_code()

except ValueError:
    print("Failed")

else:
    print("Success")
```

With `finally`

```
try:
    risky_code()

except ValueError:
    print("Failed")

finally:
    print("Finished")
```

---

40. Key Takeaways

- `try` contains code that might raise an exception.
- `except` handles the exception.
- Python skips the remaining `try` code after an exception occurs.

- Use specific exception types whenever possible.
- Multiple `except` blocks can handle different problems.
- Multiple exception types can share one handler.
- `as` lets you access the exception object.
- `else` runs only when no exception occurs.
- `finally` can be used for code that must run afterward.
- Keep `try` blocks focused.
- Do not silently ignore exceptions.
- Exception handling is especially useful for user input, files, calculations, APIs, and other operations where failures are possible.
- Good exception handling makes programs more reliable without hiding genuine bugs.

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

Visit haas.dev for more resources and guides.