Python else and finally

When working with exceptions, Python gives us more control than just try and except.

Two important keywords are:

- else
- finally

They help us decide **when code should run after try**, and **which code should always run whether an error happens or not**.

---

1. The Basic Structure

The complete exception-handling structure can look like this:

```
try:
    # code that might cause an error
except:
    # runs if an error occurs
else:
    # runs if no error occurs
finally:
    # always runs
```

Think of it like this:

```
TRY
 
 □□□ Error □□□ EXCEPT
 
 □□□ No Error □□□ ELSE
```

```
        □
        □
    FINALLY
```

The important idea is:

> `else` runs only when `try` succeeds, while `finally` runs whether `try` succeeds or fails.

---

2. What Is `else` in Exception Handling?

The `else` block runs **only when the code inside `try` executes successfully**.

Example:

```
try:
    number = int("50")
except ValueError:
    print("Invalid number")
else:
    print("Conversion successful")
```

Output:

```
Conversion successful
```

There was no error, so `else` ran.

---

3. What Happens When an Error Occurs?

Consider:

```
try:
    number = int("hello")
except ValueError:
    print("Invalid number")
else:
    print("Conversion successful")
```

Output:

```
Invalid number
```

The `try` block failed.

Therefore:

- `except` runs
- `else` does not run

---

4. Why Use `else`?

You could technically put everything inside `try`, but that can make error handling too broad.

For example:

```
try:
    number = int(input("Enter a number: "))
    result = number * 2
    print(result)
except ValueError:
```

```
    print("Invalid input")
```

The entire block is being monitored for `ValueError`.

A cleaner approach is:

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Invalid input")
else:
    result = number * 2
    print(result)
```

Now the `try` block contains only the operation that can realistically raise the expected error.

The `else` block contains code that should run **after successful validation/conversion**.

---

5. A Simple Real Life Example

Imagine entering your age into a form.

```
try:
    age = int(input("Enter your age: "))
except ValueError:
    print("Please enter a valid number")
else:
    print(f"You are {age} years old")
```

If the user enters:

```
20
```

Output:

```
You are 20 years old
```

If the user enters:

```
twenty
```

Output:

```
Please enter a valid number
```

---

6. `else` Does Not Mean "No Exception Anywhere"

It specifically means:

> Run this block if the `try` block completed without raising an exception.

Example:

```
try:
    number = int("10")
except ValueError:
    print("Invalid")
else:
    print("Everything in try worked")
```

Output:

```
Everything in try worked
```

---

7. What Is `finally`?

The `finally` block is different.

It runs **regardless of whether an exception occurs**.

Example:

```
try:
    print("Trying")
except:
    print("Error")
finally:
    print("Finished")
```

Output:

```
Trying
Finished
```

---

8. `finally` When an Error Happens

```
try:
    number = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("This always runs")
```

Output:

```
Cannot divide by zero
This always runs
```

The error occurred, `except` handled it, and then `finally` still executed.

---

9. `finally` When No Error Happens

```
try:
    result = 10 / 2
    print(result)
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Operation finished")
```

Output:

```
5.0
Operation finished
```

No error occurred, but `finally` still ran.

---

10. The Main Difference

Keyword	When does it run?

try	Attempts the code
except	When a matching exception occurs
else	When no exception occurs
finally	Almost always, regardless of success or error

Easy way to remember:

try Try this
except Something went wrong
else Everything worked
finally Do this at the end

---

11. Using All Four Together

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")
else:
```

```
    print(f"Result: {result}")
finally:
    print("Program finished")
```

User enters `10`

```
Result: 10.0
Program finished
```

User enters `0`

```
Number cannot be zero
Program finished
```

User enters `hello`

```
Please enter a valid number
Program finished
```

Notice:

- `else` runs only for `10`
- `finally` runs in all three cases

---

12. Execution Order

Suppose this code runs successfully:

```
try:
    print("1")
except:
    print("2")
else:
```

```
    print("3")
finally:
    print("4")
```

Output:

```
1
3
4
```

If an exception occurs:

```
try:
    print("1")
    10 / 0
except ZeroDivisionError:
    print("2")
else:
    print("3")
finally:
    print("4")
```

Output:

```
1
2
4
```

So the flow is:

Successful `try`

try → else → finally

Failed `try`

`try` → `except` → `finally`

---

13. `finally` Is Useful for Cleanup

One of the biggest purposes of `finally` is **cleanup**.

Some operations require resources that should be released after use.

Examples:

- files
- database connections
- network connections
- temporary resources

Example:

```
file = None

try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found")
finally:
    if file:
        file.close()
```

Whether reading succeeds or fails, Python reaches the `finally` block.

---

14. `finally` and Files

A simple example:

```
file = open("data.txt", "r")

try:
    data = file.read()
    print(data)
finally:
    file.close()
```

The important idea is:

```
finally:
    file.close()
```

The file should be closed after the operation.

However, in modern Python, there is an even better way to handle files:

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

The `with` statement automatically handles cleanup.

This leads to an important Python concept called **context managers**, which can be studied separately.

---

15. `finally` Runs Even After `return`

Consider:

```
def example():
    try:
        return "Done"
    finally:
        print("Finally runs")

print(example())
```

Output:

```
Finally runs
Done
```

Even though `return` was reached, the `finally` block ran first.

This is an important behavior to understand.

---

16. `finally` Can Run After `except`

```
def divide():
    try:
        return 10 / 0
    except ZeroDivisionError:
        return 0
    finally:
        print("Cleanup")
```

```
print(divide())
```

Output:

```
Cleanup
0
```

The `finally` block executes before the function actually returns its result.

---

17. Avoid Putting Important Logic in `finally`

Although `finally` is powerful, it should usually be used for cleanup.

Good:

```
try:
    connection = connect()
finally:
    connection.close()
```

Less appropriate:

```
finally:
    print("User has successfully purchased the product")
```

Why?

Because `finally` runs even when something fails.

The purchase-success message belongs in `else` or normal successful logic, not `finally`.

---

18. `else` vs `finally`

Consider this example:

```
try:
    payment = process_payment()
except PaymentError:
    print("Payment failed")
else:
    print("Payment successful")
finally:
    print("Payment process ended")
```

The roles are very clear:

`try`

Attempts the payment.

`except`

Handles a payment failure.

`else`

Runs only if payment succeeds.

`finally`

Runs after the payment attempt regardless of the result.

---

19. A Practical Example

Imagine a login system:

```
try:
    username = input("Username: ")
    password = input("Password: ")

    if password == "":
        raise ValueError("Password cannot be empty")

except ValueError as error:
    print(error)

else:
    print(f"Login attempt for {username}")

finally:
    print("Login process completed")
```

Possible output when password is provided:

```
Login attempt for Hafsa
Login process completed
```

If password is empty:

```
Password cannot be empty
Login process completed
```

---

20. `else` Helps Keep `try` Focused

A common mistake is putting too much code inside `try`.

Instead of:

```
try:
    number = int(input("Number: "))
    result = number * 2
    print(result)
    save_result(result)
except ValueError:
    print("Invalid input")
```

Prefer:

```
try:
    number = int(input("Number: "))
except ValueError:
    print("Invalid input")
else:
    result = number * 2
    print(result)
    save_result(result)
```

The `try` block now focuses on the operation that can produce the expected `ValueError`.

This makes programs easier to understand and debug.

---

21. `else` Is Not the Same as `if`

Do not confuse:

```
try:
    number = int(input("Number: "))
except ValueError:
    print("Invalid")
else:
    print("Valid")
```

with:

```
number = int(input("Number: "))

if number:
    print("Valid")
```

`else` in exception handling is connected to whether an **exception occurred**.

It does not test whether a value is `True` or `False`.

---

22. `finally` Is Not the Same as Normal Code After `try`

Consider:

```
try:
    print("Working")
except:
    print("Error")

print("Finished")
```

Usually, `"Finished"` runs after the exception is handled.

But `finally` has a special purpose:

```
try:
    print("Working")
except:
    print("Error")
finally:
    print("Finished")
```

`finally` is specifically designed for code that must run as part of the exception-handling flow.

It is especially useful for cleanup.

---

23. Common Mistakes

Mistake 1: Thinking `else` runs when an error occurs

Wrong:

```
try:
    10 / 0
except ZeroDivisionError:
    print("Error")
else:
    print("Success")
```

Output:

```
Error
```

`else` does not run.

Mistake 2: Thinking `finally` only runs after errors

Wrong assumption:

`finally = error happened`

Correct:

`finally = final cleanup code`

It normally runs whether there is an error or not.

Mistake 3: Putting everything inside `try`

Avoid:

```
try:
    # 30 lines of unrelated code
except Exception:
    print("Something went wrong")
```

Keep the `try` block focused.

Mistake 4: Using `finally` for success messages

Avoid:

`finally:`

```
    print("Payment successful")
```

The code runs even if payment failed.

Use:

```
else:
    print("Payment successful")
```

instead.

Mistake 5: Using a broad `except` unnecessarily

Avoid:

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

Prefer:

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

Specific exceptions make programs easier to understand and debug.

---

24. Best Practices

1. Keep `try` small

Only put operations that may realistically raise the expected exception inside it.

2. Use `else` for successful follow-up logic

If an operation succeeds, `else` can handle what should happen next.

3. Use `finally` for cleanup

Good examples:

`file.close()`

`connection.close()`

`release_resource()`

4. Catch specific exceptions

Prefer:

`except ValueError:`

over:

`except:`

5. Don't hide unexpected errors

Exception handling should make problems easier to understand, not silently hide them.

---

25. A Mental Model

Imagine ordering food online.

```
TRY
□
Place the order
□
□□□ Problem?
□     □
□   EXCEPT
□   Show error
□
□□□ No problem?
      □
    ELSE
    Show order confirmation

FINALLY
□
Finish the order process
```

This mental model makes the difference easy:

> `else` = success path
> `finally` = always-run cleanup/finalization path

---

26. `try`, `except`, `else`, `finally` Cheat Sheet

```
try:
    risky_operation()

except SpecificError:
    handle_error()

else:
    successful_operation()

finally:
    cleanup()
```

Remember:

```
try      → Try something
except   → Handle failure
else     → Handle success
finally  → Clean up / finish
```

---

27. Problem Solving Framework

When deciding where code belongs, ask:

Question 1

Can this operation raise the exception I'm handling?

If yes:

```
try:
```

Question 2

What should happen if that specific exception occurs?

Put it in:

`except:`

Question 3

What should happen only if everything worked?

Put it in:

`else:`

Question 4

What should happen regardless of success or failure?

Put it in:

`finally:`

---

28. Mini Project: Safe Number Processor

Build a small program that:

1. Asks the user for a number.
2. Converts it to an integer.
3. Divides 100 by that number.
4. Handles invalid input.
5. Displays the result only when successful.

6. Always prints a completion message.

Solution:

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

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

except ZeroDivisionError:
    print("You cannot enter zero")

else:
    print(f"Result: {result}")

finally:
    print("Program completed")
```

---

29. 5 Minute Challenge

Create a program that asks for a user's age.

Requirements:

- Convert the input into an integer.
- Handle invalid input using `except`.
- Use `else` to print the age when conversion succeeds.
- Use `finally` to print:

```
Age check completed
```

Example:

```
Enter your age: 20
Your age is 20
Age check completed
```

---

30. Exercises

Exercise 1

Predict the output:

```
try:
    print("A")
except:
    print("B")
else:
    print("C")
finally:
    print("D")
```

Exercise 2

Predict the output:

```
try:
    number = 10 / 0
except ZeroDivisionError:
    print("Error")
else:
```

```
    print("Success")
finally:
    print("Done")
```

Exercise 3

Create a program that:

- asks for two numbers
- divides the first by the second
- handles invalid input
- handles division by zero
- uses `else` for the successful result
- uses `finally` for a completion message

Exercise 4

Explain in your own words:

> Why does `else` not run when an exception occurs?

Exercise 5

Create a file-reading program using:

```
try
except
```

`finally`

Make sure the file is closed in `finally`.

---

31. Mini Quiz

1. When does `else` execute?

A. Always
B. Only when `try` succeeds
C. Only when an exception occurs
D. Before `try`

Answer: B

---

2. When does `finally` normally execute?

A. Only when an error occurs
B. Only when `try` succeeds
C. Whether an exception occurs or not
D. Only when `except` exists

Answer: C

---

3. Which block is commonly used for cleanup?

A. `try`

B. except
C. else
D. finally

Answer: D

---

4. Which flow represents a successful try?

A.

try → except → finally

B.

try → else → finally

C.

try → finally → else

D.

else → try → finally

Answer: B

---

5. Which flow represents a handled exception?

A.

`try → else → finally`

B.

`try → except → finally`

C.

`except → try → finally`

D.

`try → finally → except`

Answer: B

---

32. Interview Questions

Beginner

1. What is the purpose of `else` with `try`?
2. What is the purpose of `finally`?
3. When does `else` execute?
4. When does `finally` execute?
5. Can `try` be used without `except`?

Intermediate

6. What is the difference between `else` and `finally`?
7. Why should the `try` block usually be kept small?
8. Why is `finally` useful for resource cleanup?
9. Does `finally` execute if the `try` block contains `return`?

10. When should you use `else` instead of putting code directly inside `try`?

Practical

11. How would you safely handle file cleanup?
12. How would you structure a database operation using `try`, `except`, `else`, and `finally`?
13. Why is a success message usually better inside `else` than `finally`?
14. What happens if an exception occurs inside the `else` block?
15. What happens if the `finally` block itself raises an exception?

---

33. Quick Comparison

Feature	try	except	else	finally
Main purpose	Attempt code	Handle errors	Handle success	Finalize/cleanup
Runs on success	Yes	No	Yes	Yes
Runs on handled error	Stops at error	Yes	No	Yes
Common use	Risky operation	Error handling	Successful follow-up	Cleanup
Example	`int(value)`	`except ValueError`	Show result	Close resource

---

34. Key Takeaways

- `else` runs when the `try` block completes successfully.
- `else` does not run when the `try` block raises an exception.
- `finally` is designed for code that should run at the end of the exception-handling process.
- `finally` normally runs whether an exception occurs or not.
- `else` is useful for separating successful logic from risky operations.
- `finally` is commonly used for cleanup.
- Keep `try` blocks small and focused.
- Catch specific exceptions whenever possible.
- Do not use `finally` for logic that should happen only after success.

The core pattern to remember is:

```
try:
    # risky operation

except SomeError:
    # handle failure

else:
    # handle success

finally:
    # cleanup / final step
```

Once you understand this pattern, you can build more reliable Python programs that handle errors without making the rest of your code difficult to manage.

Visit haas.dev for more resources and guides.

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

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