Python Errors and Exceptions

Introduction

While writing Python programs, things can go wrong.

You might:

- write incorrect syntax
- use a variable that does not exist
- divide by zero
- open a file that is missing
- convert invalid text into a number
- receive unexpected data from a user

Python reports these problems using **errors and exceptions**.

Understanding them is important because real programs must not only work when everything goes perfectly. They must also handle unexpected situations safely.

---

1. What Is an Error?

An **error** is a problem in a program that prevents it from working as expected.

For example:

```
print("Hello"
```

This code is missing a closing parenthesis.

Python reports:

```
SyntaxError
```

Errors help developers identify what went wrong.

---

2. Two Important Categories

Python problems can broadly be understood as:

```
Python Problems
□
□□□ Syntax Errors
□
□□□ Exceptions
```

Syntax Error

The code does not follow Python's syntax rules.

Exception

The syntax is valid, but something goes wrong while the program is running.

Understanding this difference is important.

---

3. Syntax Errors

A **SyntaxError** happens when Python cannot understand the structure of your code.

Example:

```
if age >= 18
    print("Adult")
```

The colon is missing.

Python reports a syntax error.

Correct version:

```
if age >= 18:
    print("Adult")
```

Another example:

```
print("Hello"
```

Correct:

```
print("Hello")
```

Syntax errors normally need to be fixed in the code itself.

---

4. Exceptions

An **exception** occurs while Python is executing valid code.

Example:

```
number = 10
result = number / 0
```

Python understands the syntax, but division by zero is not allowed.

Python raises:

```
ZeroDivisionError
```

Another example:

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

Python cannot convert `"hello"` into an integer.

It raises:

```
ValueError
```

---

5. Common Python Exceptions

Some common exceptions include:

```
ValueError
TypeError
NameError
IndexError
KeyError
ZeroDivisionError
FileNotFoundError
AttributeError
```

`ImportError`
`ModuleNotFoundError`

Learning what they mean makes debugging much easier.

---

6. ValueError

A `ValueError` occurs when a value has the correct general type but an inappropriate value.

Example:

```
age = int("hello")
```

The value cannot be converted to an integer.

Another example:

```
numbers = [1, 2, 3]

numbers.remove(10)
```

The value `10` does not exist in the list.

---

7. TypeError

A `TypeError` happens when an operation is used with an inappropriate data type.

Example:

```
result = "10" + 5
```

Python cannot directly add a string and an integer.

Correct:

```
result = int("10") + 5
```

Another example:

```
len(10)
```

An integer does not have a length.

---

8. NameError

A `NameError` occurs when Python cannot find a variable or name.

Example:

```
print(username)
```

If `username` was never defined, Python raises:

```
NameError
```

Correct:

```
username = "Hafsa"

print(username)
```

---

9. IndexError

An `IndexError` occurs when you try to access a list position that does not exist.

Example:

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

print(numbers[5])
```

The valid indexes are:

```
0
1
2
```

There is no index `5`.

Correct:

```
print(numbers[2])
```

---

10. KeyError

A `KeyError` commonly occurs when you try to access a dictionary key that does not exist.

Example:

```
user = {
```

```
    "name": "Hafsa",
    "age": 25
}

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

There is no `"email"` key.

A safer approach is:

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

This returns:

```
None
```

instead of raising a `KeyError`.

---

11. ZeroDivisionError

This occurs when you attempt to divide by zero.

```
result = 10 / 0
```

Python raises:

```
ZeroDivisionError
```

You can prevent it by checking first:

```
number = 0
```

```
if number != 0:
    result = 10 / number
```

Or handle the exception:

```
try:
    result = 10 / number
except ZeroDivisionError:
    print("Cannot divide by zero")
```

---

12. FileNotFoundError

This occurs when Python tries to open a file that does not exist.

```
file = open("missing.txt")
```

If the file is unavailable, Python can raise:

```
FileNotFoundError
```

Later, exception handling can be used to handle this situation gracefully.

---

13. What Is Exception Handling?

Exception handling means writing code that can respond to unexpected situations instead of allowing the program to crash immediately.

Python mainly uses:

`try`
`except`
`else`
`finally`

The basic structure is:

```
try:
    # code that may cause an exception
except:
    # code that handles the exception
```

---

14. The `try` Block

The `try` block contains code that might raise an exception.

Example:

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

Python attempts to execute the code inside `try`.

If nothing goes wrong, execution continues normally.

If an exception occurs, Python looks for a matching `except` block.

---

15. The `except` Block

The `except` block handles an exception.

Example:

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

If the user enters:

```
abc
```

the program does not immediately crash.

Instead:

```
Please enter a valid number.
```

---

16. Handling Specific Exceptions

It is usually better to handle the specific exception you expect.

Good:

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

This tells Python exactly what problem you are handling.

---

17. Handling Multiple Exceptions

You can have multiple `except` blocks.

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.")
```

Different problems can receive different responses.

---

18. Multiple Exceptions With One `except`

If different exceptions should receive the same response, they can be grouped.

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

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

This is useful when the handling logic is identical.

---

19. The `else` Block

Python also provides `else`.

The `else` block runs **only when no exception occurs**.

Example:

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Invalid number.")
else:
    print("You entered:", number)
```

Flow:

```
try
 □
 □□□ Exception □ except
 □
 □□□ No exception □ else
```

---

20. The `finally` Block

The `finally` block runs whether an exception occurs or not.

Example:

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Invalid number.")
finally:
    print("Program finished.")
```

If the input is valid:

```
Program finished.
```

If the input is invalid:

```
Invalid number.
Program finished.
```

`finally` is useful for cleanup operations.

---

21. Complete Exception Handling Structure

Python allows all four parts:

```
try:
    # risky code

except ValueError:
    # handle error

else:
    # runs if no exception

finally:
```

```
    # always runs
```

Not every situation requires all four.

---

22. A Real Example: User Input

Without exception handling:

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

print(age)
```

If the user enters:

```
twenty
```

the program crashes with `ValueError`.

With exception handling:

```
try:
    age = int(input("Enter your age: "))
    print("Your age is:", age)

except ValueError:
    print("Please enter your age as a number.")
```

Now the program responds properly to invalid input.

---

23. Exception Flow

Consider:

```
try:
    number = int("abc")
    print("This will not run")

except ValueError:
    print("Invalid conversion")

print("Program continues")
```

Flow:

```
try starts
   
int("abc")
   
ValueError
   
except runs
   
Program continues
```

The line after the failing statement inside `try` is skipped.

---

24. Catching Too Broadly

Python allows:

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

This catches almost any exception.

However, it is often better to catch specific exceptions:

```
try:
    risky_code()
except ValueError:
    print("Invalid value")
```

Specific exceptions make programs easier to understand and debug.

---

25. The `Exception` Class

You can catch many ordinary exceptions using:

```
except Exception as error:
```

Example:

```
try:
    result = 10 / 0

except Exception as error:
    print("An error occurred:", error)
```

Output will describe the exception.

This can be useful when you need to log unexpected problems.

However, it should not replace thoughtful exception handling.

---

26. Reading the Exception Message

Example:

```
try:
    number = int("hello")
except ValueError as error:
    print(error)
```

Python provides information about the problem.

You can use the exception object:

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

This is useful for debugging and logging.

---

27. Raising an Exception

You can also intentionally create an exception using:

```
raise
```

Example:

```
age = -5

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

Python raises:

```
ValueError
```

with the custom message.

---

28. Why Use `raise`?

Sometimes your function receives invalid data and should stop immediately.

Example:

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

    return age
```

Now:

```
set_age(-10)
```

raises an exception.

This allows the function to protect its own rules.

---

29. Re-Raising an Exception

Sometimes you catch an exception to perform some work and then want to let it continue upward.

You can use:

```
raise
```

Example:

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

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

The exception is handled temporarily and then raised again.

This is useful in larger applications when an error needs to be logged before another part of the program handles it.

---

30. Custom Exceptions

Python allows you to create your own exception classes.

Example:

```
class InvalidAgeError(Exception):
    pass
```

Now:

```
age = -5

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

You can handle it:

```
try:
    age = -5

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

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

Custom exceptions are especially useful in larger applications.

---

31. Exception Hierarchy

Python exceptions are organized into a hierarchy.

A simplified view:

```
BaseException
□
□□□ Exception
□   □
□   □□□ ValueError
```

□ □□□ TypeError
□ □□□ IndexError
□ □□□ KeyError
□ □□□ OSError
□ □ □□□ FileNotFoundError
□ □□□ ...
□
□□□ SystemExit / KeyboardInterrupt / ...

This explains why:

`except Exception:`

can catch many ordinary application exceptions.

---

32. Errors vs Exceptions

A useful distinction:

Concept	Meaning
Syntax Error	Python cannot understand the code structure
Exception	A problem occurs while the program is running
Exception Handling	Code that responds to runtime problems

`raise`	Manually creates an exception
Custom Exception	An exception designed for your application's rules

---

33. Errors Are Not Always Programmer Mistakes

Some exceptions are expected possibilities.

For example:

```
try:
    age = int(input("Age: "))
except ValueError:
    print("Invalid input")
```

A user entering letters instead of numbers does not necessarily mean the program itself is broken.

The program should be designed to handle this possibility.

This is an important mindset:

> **Good software expects some things to go wrong.**

---

34. Preventing vs Handling Exceptions

There are two useful approaches.

Prevent the problem

```
if number != 0:
    result = 100 / number
```

Handle the problem

```
try:
    result = 100 / number
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Both approaches can be useful.

A good program uses validation and exception handling where appropriate.

---

35. Validation Before Processing

Suppose a user enters an age.

You can validate it:

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

if age < 0:
    print("Age cannot be negative.")
```

For conversion itself, exception handling is still useful:

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

    if age < 0:
        print("Age cannot be negative.")

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

Here:

```
Conversion → exception handling
Business rule → validation
```

These are related but different responsibilities.

---

36. File Example

Suppose you want to read a file:

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

except FileNotFoundError:
    print("The file does not exist.")
```

The program can respond gracefully instead of crashing.

This becomes especially useful when working with user files, configuration files, or external data.

---

37. Exception Handling With Functions

Exception handling can be placed inside a function.

```
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)
```

The function hides the input-conversion details from the rest of the program.

---

38. Exception Handling With Modules

Exception handling also works naturally with your own modules.

calculator.py

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

```
    return a / b
```

main.py

```
import calculator

try:
    result = calculator.divide(10, 0)
    print(result)

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

The module defines the rule.

The calling code decides how to respond.

This separation is useful in larger applications.

---

39. Real World Example: Resource Search

Imagine a resource system that expects a search query.

```
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 function protects itself from invalid input.

The calling code can handle the exception:

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

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

This is how exception handling becomes part of application design.

---

40. Logging Errors

In larger applications, simply printing an error may not be enough.

Developers often **log** exceptions so they can investigate them later.

A simple example:

```
import logging

logging.basicConfig(level=logging.ERROR)

try:
    result = 10 / 0

except ZeroDivisionError:
    logging.exception("Division failed")
```

Logging is especially useful for:

- web applications
- APIs
- background jobs
- production software
- debugging unexpected failures

---

41. Common Mistakes

Mistake 1: Catching Everything

Avoid:

```
try:
    ...
except:
    pass
```

This can hide important problems.

Mistake 2: Silently Ignoring Exceptions

Bad:

```
try:
    process_data()
except Exception:
    pass
```

Now you may never know that something failed.

Mistake 3: Using Exceptions Instead of Normal Logic

Do not intentionally create exceptions for ordinary decisions when a simple condition is clearer.

Prefer:

```
if age >= 18:
    print("Adult")
```

rather than deliberately causing an exception and catching it.

Mistake 4: Giving Unhelpful Messages

Avoid:

```
print("Error")
```

Prefer:

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

A useful message tells the user what they can do next.

Mistake 5: Catching the Wrong Exception

If you expect:

```
ValueError
```

handle:

`except ValueError:`

rather than catching every possible exception.

---

42. Best Practices

Catch specific exceptions

`except ValueError:`

is usually better than:

`except:`

Keep the `try` block focused

Avoid putting the entire program inside one `try` block.

Instead:

```
try:
    number = int(user_input)
except ValueError:
    ...
```

Give useful messages

Tell users what went wrong and, when appropriate, how to fix it.

Do not hide errors

Avoid:

```
except:
    pass
```

unless there is a very specific reason.

Use `finally` for cleanup

Use it when something must happen whether an operation succeeds or fails.

Use custom exceptions when appropriate

They can make large applications easier to understand.

---

43. Exception Handling Workflow

When writing code that can fail, think:

```
1. What can go wrong?
        
2. Which exception represents it?
        
3. Can I prevent it?
        
4. If not, how should I handle it?
        
5. What should the user see?
        
6. Should the error be logged?
```

This is a practical way to design reliable programs.

---

44. Mini Project: Safe Calculator

Create:

safe_calculator.py

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

    return a / b
```

Then:

main.py

```
from safe_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)
```

```
finally:
    print("Calculator finished.")
```

This small project demonstrates:

`try`
`except`
`else`
`finally`
`raise`

---

45. Five Minute Challenge

Create a function:

```
def calculate_average(numbers):
    ...
```

It should:

1. Accept a list of numbers.
2. Raise `ValueError` if the list is empty.
3. Return the average.
4. Handle the exception when the function is called.

Example:

```
calculate_average([10, 20, 30])
```

should return:

```
20.0
```

But:

```
calculate_average([])
```

should raise a meaningful exception.

---

46. Exercises

Exercise 1

Write a program that asks for two numbers and handles:

- `ValueError`
- `ZeroDivisionError`

Exercise 2

Create a function:

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

Handle an invalid list index using `IndexError`.

Exercise 3

Create a dictionary and safely access a key.

Practice handling:

`KeyError`

Exercise 4

Write a program that opens a file and handles:

`FileNotFoundError`

Exercise 5

Create:

```
class InvalidScoreError(Exception):
    pass
```

Raise it when a score is below `0` or above `100`.

---

47. Mini Quiz

Question 1

What is a `SyntaxError`?

A. A problem with a database

B. Invalid Python syntax

C. A missing file

D. A wrong dictionary key

Answer: B

Question 2

Which block contains code that might raise an exception?

A. `try`

B. `except`

C. `else`

D. `finally`

Answer: A

Question 3

Which exception occurs when converting `"abc"` to an integer?

```
int("abc")
```

Answer: `ValueError`

Question 4

Which exception occurs when dividing by zero?

Answer: `ZeroDivisionError`

Question 5

Which block runs when no exception occurs?

Answer: `else`

Question 6

Which block normally runs whether an exception occurs or not?

Answer: `finally`

Question 7

How do you intentionally raise an exception?

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

---

48. Interview Questions

1. What is the difference between an error and an exception?

A syntax error prevents Python from understanding the code, while an exception generally occurs during execution when something unexpected happens.

2. What is exception handling?

It is the process of detecting and responding to runtime exceptions using constructs such as `try` and `except`.

3. What is the purpose of `try`?

It contains code that may raise an exception.

4. What is the purpose of `except`?

It handles a matching exception.

5. What is the purpose of `else`?

It runs when the `try` block completes without an exception.

6. What is the purpose of `finally`?

It runs whether an exception occurs or not.

7. What does `raise` do?

It explicitly raises an exception.

8. Why should you avoid `except:` when possible?

It can catch unexpected problems and make debugging more difficult.

9. What is a custom exception?

A programmer-defined exception class used to represent application-specific problems.

10. What is the difference between `ValueError` and `TypeError`?

`ValueError` usually means the type is acceptable but the value is inappropriate. `TypeError` usually means an operation received an inappropriate type.

---

49. Cheat Sheet

Basic handling

```
try:
    risky_code()

except ValueError:
    handle_error()
```

Multiple exceptions

```
try:
    risky_code()

except ValueError:
    ...

except TypeError:
    ...
```

`else`

```
try:
```

```
    risky_code()

except ValueError:
    ...

else:
    success_code()
```

finally

```
try:
    risky_code()

except ValueError:
    ...

finally:
    cleanup()
```

Raise an exception

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

Custom exception

```
class MyError(Exception):
    pass
```

Exception object

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

---

50. Key Takeaways

- Errors indicate problems in a Python program.
- Syntax errors happen when Python cannot understand the code structure.
- Exceptions usually happen during program execution.
- Common exceptions include `ValueError`, `TypeError`, `NameError`, `IndexError`, `KeyError`, and `ZeroDivisionError`.
- `try` contains code that might fail.
- `except` handles an exception.
- `else` runs when no exception occurs.
- `finally` runs regardless of whether an exception occurs.
- `raise` allows you to intentionally raise an exception.
- Custom exceptions allow applications to define meaningful error types.
- Specific exception handling is generally better than catching everything.
- Exception handling makes programs more reliable and user-friendly.
- Good developers think about what can go wrong before the program reaches production.

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

Visit haas.dev for more resources and guides.