Python Custom Exceptions

Python already provides many built in exceptions:

`ValueError`
`TypeError`
`KeyError`
`IndexError`
`FileNotFoundError`
`ZeroDivisionError`

These are useful for common programming problems.

But as applications become larger, we often need to describe **our own specific errors**.

For example:

`InsufficientBalanceError`
`InvalidUsernameError`
`ResourceNotFoundError`
`AuthenticationError`
`PaymentFailedError`

These are called **custom exceptions**.

A custom exception is simply an exception class that we create for our own application's rules and problems.

---

1. Why Do We Need Custom Exceptions?

Imagine a banking application.

We could write:

```
raise ValueError("Insufficient balance")
```

This works.

But `ValueError` is very general.

A developer reading:

```
except ValueError:
```

cannot immediately know whether the problem was:

- insufficient balance
- invalid age
- invalid username
- invalid price
- invalid quantity

A custom exception makes the meaning much clearer:

```
raise InsufficientBalanceError("Not enough money")
```

Now the exception itself communicates what happened.

---

2. What Is a Custom Exception?

A custom exception is a Python class that inherits from an existing exception class.

The simplest example:

```
class InsufficientBalanceError(Exception):
    pass
```

Now we have created our own exception:

```
InsufficientBalanceError
```

It behaves like other Python exceptions.

We can raise it:

```
raise InsufficientBalanceError("Not enough balance")
```

---

3. The Basic Pattern

The most common pattern is:

```
class MyCustomError(Exception):
    pass
```

Then:

```
raise MyCustomError("Something went wrong")
```

And handle it:

```
try:
    raise MyCustomError("Something went wrong")
```

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

Output:

```
Something went wrong
```

---

4. Why Inherit From `Exception`?

Python's exception system is based on classes.

The built in `Exception` class is the standard base class for most application exceptions.

When we write:

```
class InsufficientBalanceError(Exception):
    pass
```

we are saying:

> `InsufficientBalanceError` is a type of Python exception.

This allows Python to use it with:

```
raise
```

and:

```
except
```

---

5. Creating Your First Custom Exception

Let's create an exception for invalid usernames.

```
class InvalidUsernameError(Exception):
    pass
```

Now use it:

```
username = ""

if not username:
    raise InvalidUsernameError("Username cannot be empty")
```

Python reports:

```
InvalidUsernameError: Username cannot be empty
```

The name itself tells us what went wrong.

---

6. Handling a Custom Exception

We can catch it just like any built in exception.

```
class InvalidUsernameError(Exception):
    pass


try:
    username = ""
```

```
    if not username:
        raise InvalidUsernameError("Username cannot be empty")

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

Output:

```
Username cannot be empty
```

---

7. Custom Exceptions vs Built In Exceptions

Compare these two:

Built in exception

```
raise ValueError("Insufficient balance")
```

Custom exception

```
raise InsufficientBalanceError("Insufficient balance")
```

Both work.

But the second one gives the application a more specific error type.

You can now write:

```
except InsufficientBalanceError:
```

instead of:

```
except ValueError:
```

This makes the code easier to understand.

---

8. A Banking Example

Let's create a custom exception:

```
class InsufficientBalanceError(Exception):
    pass
```

Now:

```
def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientBalanceError("Insufficient balance")

    return balance - amount
```

Use it:

```
try:
    balance = withdraw(1000, 1500)
    print(balance)

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

Output:

```
Insufficient balance
```

The code clearly communicates the problem.

---

9. Multiple Custom Exceptions

A real application may need several custom exceptions.

```
class InsufficientBalanceError(Exception):
    pass


class InvalidAmountError(Exception):
    pass


class AccountNotFoundError(Exception):
    pass
```

Now different situations can have different exception types.

```
def withdraw(balance, amount):

    if amount <= 0:
        raise InvalidAmountError("Amount must be greater than zero")

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

    return balance - amount
```

The caller can handle them separately:

```
try:
    balance = withdraw(1000, 1500)

except InvalidAmountError:
    print("Invalid withdrawal amount")

except InsufficientBalanceError:
    print("You do not have enough balance")
```

---

10. Why Separate Exceptions Matter

Suppose we only use:

```
raise ValueError("Something went wrong")
```

The caller cannot easily distinguish between different problems.

With custom exceptions:

```
InvalidAmountError
        
Invalid amount

InsufficientBalanceError
        
Not enough money

AccountNotFoundError
        
Account doesn't exist
```

Different problems can now have different handling.

---

11. Custom Exceptions With Functions

Custom exceptions are particularly useful when functions enforce business rules.

Example:

```
class InvalidAgeError(Exception):
    pass


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

    return "Registration successful"
```

Use:

```
try:
    print(register_user(10))

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

Output:

```
User must be at least 13 years old
```

---

12. Custom Exceptions and Validation

Consider a registration system.

```
class InvalidUsernameError(Exception):
    pass


class InvalidAgeError(Exception):
    pass


def register(username, age):

    if not username:
        raise InvalidUsernameError("Username cannot be empty")

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

    return "Registration successful"
```

Now:

```
try:
    register("", 20)

except InvalidUsernameError as error:
    print(error)

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

Output:

```
Username cannot be empty
```

---

13. Custom Exception Hierarchies

Custom exceptions can also inherit from other custom exceptions.

For example:

```
class ApplicationError(Exception):
    pass


class PaymentError(ApplicationError):
    pass


class CardPaymentError(PaymentError):
    pass


class PaymentTimeoutError(PaymentError):
    pass
```

Now we have a hierarchy:

```
Exception
    
    ApplicationError
        
        PaymentError
            
            CardPaymentError
            
            PaymentTimeoutError
```

This allows us to catch errors at different levels.

---

14. Catching a Parent Exception

Suppose:

```
class ApplicationError(Exception):
    pass


class PaymentError(ApplicationError):
    pass
```

If we raise:

```
raise PaymentError("Payment failed")
```

we can catch it with:

```
except PaymentError:
```

or:

```
except ApplicationError:
```

because `PaymentError` inherits from `ApplicationError`.

---

15. Why Create an Exception Hierarchy?

Imagine a payment system:

```
ApplicationError
□
□□□ AuthenticationError
□
□□□ PaymentError
□   □□□ CardPaymentError
□   □□□ PaymentTimeoutError
□
□□□ ResourceNotFoundError
```

The application can handle a specific error:

```
except CardPaymentError:
```

or a broader category:

```
except PaymentError:
```

or all application-specific errors:

```
except ApplicationError:
```

This gives large applications more control.

---

16. Custom Exceptions With `pass`

If your custom exception doesn't need additional behavior:

```
class ResourceNotFoundError(Exception):
```

```
  pass
```

That's completely valid.

The class itself is useful because its **type has meaning**.

You don't always need to add methods or properties.

---

17. Custom Exception With a Message

You can simply create an instance with a message:

```
raise ResourceNotFoundError("Resource was not found")
```

Then:

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

Output:

```
Resource was not found
```

Python's base `Exception` already supports storing and displaying the message.

---

18. Custom Exception With Extra Information

Sometimes we need more information than just a message.

For example:

```
class InsufficientBalanceError(Exception):

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount

        super().__init__(
            f"Balance is {balance}, but withdrawal requested is {amount}"
        )
```

Now:

```
raise InsufficientBalanceError(500, 1000)
```

The exception contains:

```
balance = 500
amount = 1000
```

and its message explains the problem.

---

19. Accessing Custom Exception Data

```
class InsufficientBalanceError(Exception):

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount

        super().__init__(
```

```
            f"Balance is {balance}, but requested amount is {amount}"
        )
```

Handle it:

```
try:
    raise InsufficientBalanceError(500, 1000)

except InsufficientBalanceError as error:
    print(error)
    print(error.balance)
    print(error.amount)
```

Output:

```
Balance is 500, but requested amount is 1000
500
1000
```

Now the exception carries structured information.

---

20. When Do We Need Extra Data?

You don't need custom attributes for every exception.

Use them when the caller genuinely needs additional information.

For example:

```
Payment failed
```

may be enough.

But for:

```
Payment failed
transaction_id = 12345
amount = 5000
reason = "insufficient funds"
```

additional data may be useful.

---

21. Custom Exception for a Resource

Imagine haas.dev has resources stored with IDs.

```
class ResourceNotFoundError(Exception):
    pass
```

Then:

```
def find_resource(resources, resource_id):

    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    raise ResourceNotFoundError(
        f"Resource '{resource_id}' was not found"
    )
```

Use:

```
try:
    resource = find_resource(resources, "python-999")

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

Output:

```
Resource 'python-999' was not found
```

This is more expressive than:

```
raise ValueError("Not found")
```

---

22. Custom Exceptions in APIs

Custom exceptions are common in backend applications.

For example:

```
class AuthenticationError(Exception):
    pass


class ResourceNotFoundError(Exception):
    pass


class PermissionDeniedError(Exception):
    pass
```

Application logic:

```
def get_resource(user, resource_id):

    if not user:
        raise AuthenticationError("User must be logged in")

    if not user["allowed"]:
        raise PermissionDeniedError("Access denied")

    resource = find_resource(resource_id)

    if resource is None:
        raise ResourceNotFoundError("Resource not found")

    return resource
```

The API layer can translate these exceptions into appropriate responses.

The business logic doesn't need to know exactly how the HTTP response will look.

---

23. Custom Exceptions Separate Logic From Error Handling

Consider:

```
Business Logic
      
Raises meaningful exception
      
Application/API layer
      
Handles exception
      
User receives response
```

For example:

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

The function's responsibility is:

> Detect the problem.

The higher-level application can decide:

> How should this problem be presented?

This separation becomes valuable as applications grow.

---

24. Custom Exceptions in Classes

Custom exceptions can also be used with classes.

```
class InsufficientBalanceError(Exception):
    pass


class BankAccount:

    def __init__(self, balance):
        self.balance = balance
```

```
    def withdraw(self, amount):

        if amount > self.balance:
            raise InsufficientBalanceError(
                "Insufficient balance"
            )

        self.balance -= amount
```

Use:

```
account = BankAccount(1000)

try:
    account.withdraw(1500)

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

Output:

```
Insufficient balance
```

---

25. Custom Exceptions and finally

Custom exceptions work normally with the complete exception-handling structure.

```
class PaymentError(Exception):
    pass


try:
```

```
    raise PaymentError("Payment failed")

except PaymentError as error:
    print(error)

else:
    print("Payment successful")

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

Output:

```
Payment failed
Payment process completed
```

The `else` block doesn't run because an exception occurred.

The `finally` block still runs.

---

26. Custom Exceptions and Re-Raising

A lower-level function may catch one exception and convert it into a custom exception.

```
class UserDataError(Exception):
    pass


def load_user():
    try:
        data = int("invalid")
```

```
    except ValueError as error:
        raise UserDataError(
            "Could not load user data"
        ) from error
```

Now the caller deals with:

```
try:
    load_user()

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

The implementation detail (`ValueError`) stays inside the lower-level function.

The application sees the meaningful domain-level exception:

```
UserDataError
```

---

27. Custom Exceptions Should Have Clear Names

Good names:

```
class InvalidUsernameError(Exception):
    pass

class PaymentFailedError(Exception):
    pass

class ResourceNotFoundError(Exception):
    pass
```

Avoid vague names:

```
class MyError(Exception):
    pass

class Problem(Exception):
    pass
```

A good exception name should tell you **what kind of problem occurred**.

---

28. Exception Naming Convention

Python convention is to end exception class names with:

```
Error
```

Examples:

```
InvalidEmailError
AuthenticationError
DatabaseConnectionError
PaymentError
ResourceNotFoundError
```

This makes it immediately obvious that the class represents an exception.

---

29. Custom Exceptions vs Many `ValueError`s

Imagine this:

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

The messages differ, but the exception type is always:

```
ValueError
```

Now imagine:

```
raise InvalidUsernameError("Username is invalid")
raise InvalidEmailError("Email is invalid")
raise InvalidPasswordError("Password is invalid")
```

The caller can handle them separately:

```
except InvalidUsernameError:
    ...

except InvalidEmailError:
    ...

except InvalidPasswordError:
    ...
```

This is one of the main benefits of custom exceptions.

---

30. Don't Create Custom Exceptions for Everything

Custom exceptions are useful, but you don't need one for every small error.

If a standard exception already clearly represents the problem, use it.

For example:

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

is perfectly reasonable.

You don't necessarily need:

```
class NegativeAgeError(Exception):
    pass
```

Use a custom exception when the error represents an important **domain-specific concept** that the application may need to identify or handle separately.

---

31. A Practical Decision Framework

Before creating a custom exception, ask:

Question 1

Does Python already have an appropriate exception?

If yes, consider using it.

Question 2

Does the error represent a meaningful application-specific concept?

If yes, a custom exception may help.

Question 3

Will the caller need to handle this error differently?

If yes, a custom exception can make that easy.

Question 4

Would the custom exception make the code clearer?

If yes, create it.

---

32. Example: E-Commerce System

Suppose an online store has these problems:

```
Product not found
Out of stock
Invalid quantity
Payment failed
```

We can model them as:

```
class StoreError(Exception):
    pass


class ProductNotFoundError(StoreError):
    pass
```

```
class OutOfStockError(StoreError):
    pass


class InvalidQuantityError(StoreError):
    pass


class PaymentFailedError(StoreError):
    pass
```

Now the application's errors have a clear structure.

---

33. Using the Exception Hierarchy

```
try:
    purchase_product()

except OutOfStockError:
    print("Product is out of stock")

except PaymentFailedError:
    print("Payment failed")

except StoreError:
    print("A store error occurred")
```

The specific errors are handled first.

The broader `StoreError` catches other store-related errors.

---

34. Important Rule: Specific Before General

Consider:

```
try:
    purchase_product()

except StoreError:
    print("Store error")

except OutOfStockError:
    print("Out of stock")
```

The second handler may never be reached because:

`OutOfStockError`
`StoreError`

`OutOfStockError` is already covered by `StoreError`.

Better:

```
try:
    purchase_product()

except OutOfStockError:
    print("Out of stock")

except StoreError:
    print("Store error")
```

General rule:

> Catch specific exceptions before their parent exceptions.

---

35. Mini Project: User Registration System

Create a custom exception hierarchy:

```
class RegistrationError(Exception):
    pass


class InvalidUsernameError(RegistrationError):
    pass


class InvalidAgeError(RegistrationError):
    pass
```

Now validation:

```
def register_user(username, age):

    if not username:
        raise InvalidUsernameError(
            "Username cannot be empty"
        )

    if age < 13:
        raise InvalidAgeError(
```

```
            "User must be at least 13 years old"
        )

    return "Registration successful"
```

Handle it:

```
try:
    result = register_user("", 20)

except InvalidUsernameError as error:
    print(error)

except InvalidAgeError as error:
    print(error)

except RegistrationError as error:
    print(error)

else:
    print(result)
```

---

36. 5 Minute Challenge

Create a custom exception:

```
class InsufficientBalanceError(Exception):
    pass
```

Then create:

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

```
    ...
```

Requirements:

- amount must be greater than zero
- amount cannot be greater than balance
- raise `ValueError` for an invalid amount
- raise `InsufficientBalanceError` when balance is insufficient
- return the remaining balance when successful

Example:

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

Expected:

```
700
```

Example:

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

Expected:

```
InsufficientBalanceError
```

---

37. Exercises

Exercise 1

Create:

```
class InvalidPasswordError(Exception):
    pass
```

Use it when a password is shorter than 8 characters.

Exercise 2

Create:

```
class ProductNotFoundError(Exception):
    pass
```

Use it when a product cannot be found in a list.

Exercise 3

Create a custom exception hierarchy:

```
ApplicationError
□□□ AuthenticationError
□□□ PaymentError
□□□ ResourceNotFoundError
```

Exercise 4

Create a function:

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

Use a custom exception for insufficient balance.

Exercise 5

Create a function:

`def find_resource(resources, resource_id):`

Raise `ResourceNotFoundError` when the ID does not exist.

---

38. Mini Quiz

1. What is a custom exception?

A. A syntax error
B. An exception class created by the programmer
C. A Python keyword
D. A package

Answer: B

---

2. What should most custom exceptions inherit from?

A. `list`
B. `dict`
C. `Exception`
D. `str`

Answer: C

---

3. Which is a good custom exception name?

A. `ErrorThing`
B. `Problem`
C. `ResourceNotFoundError`
D. `Wrong`

Answer: C

---

4. Why use custom exceptions?

A. To make Python run faster
B. To represent meaningful application-specific errors
C. To avoid functions
D. To replace all built in exceptions

Answer: B

---

5. Which should normally be caught first?

A. Parent exception
B. Specific exception
C. `Exception`
D. `BaseException`

Answer: B

---

39. Interview Questions

Beginner

1. What is a custom exception?
2. How do you create a custom exception in Python?
3. Why do custom exceptions usually inherit from `Exception`?
4. How do you raise a custom exception?
5. How do you catch a custom exception?

Intermediate

6. Why would you use a custom exception instead of `ValueError`?
7. Can custom exceptions inherit from other custom exceptions?
8. What is an exception hierarchy?
9. Why should specific exceptions be caught before general exceptions?
10. Can a custom exception contain additional attributes?

Advanced

11. What is the purpose of exception chaining with `raise ... from ...`?
12. When should you avoid creating a custom exception?
13. How can custom exceptions help separate business logic from application error handling?
14. How would you design exceptions for a payment system?
15. How would you create a base application exception with multiple specialized exceptions?

---

40. Custom Exceptions Cheat Sheet

Basic custom exception

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

Raise it

```
raise MyError("Something went wrong")
```

Catch it

```
try:
    ...
except MyError as error:
    print(error)
```

Custom hierarchy

```
class ApplicationError(Exception):
    pass


class PaymentError(ApplicationError):
    pass


class CardPaymentError(PaymentError):
    pass
```

Custom data

```
class PaymentError(Exception):

    def __init__(self, amount):
        self.amount = amount
        super().__init__(
            f"Payment failed for {amount}"
        )
```

Re-raise with a new application-level error

```
try:
    ...
except ValueError as error:
    raise PaymentError("Payment processing failed") from error
```

---

41. Key Takeaways

- A custom exception is an exception class created for your application's specific needs.
- Custom exceptions normally inherit from `Exception`.
- Use `raise` to trigger your custom exception.
- Use `except` to handle it.
- Custom exceptions make application errors more meaningful.
- They allow different types of problems to be handled differently.
- Custom exceptions can inherit from other custom exceptions to create hierarchies.
- A base application exception can provide a common category for related errors.
- Specific exceptions should be caught before their parent exceptions.
- Custom exceptions can store additional information when necessary.
- Don't create a custom exception when a standard Python exception already communicates the problem clearly.
- Use clear names ending in `Error`.
- Custom exceptions are especially useful for business rules, APIs, services, authentication, payments, databases, and other larger applications.

The core pattern is:

```
class ApplicationError(Exception):
    pass
```

```
class ResourceNotFoundError(ApplicationError):
    pass


def get_resource(resource_id):
    if not resource_exists(resource_id):
        raise ResourceNotFoundError(
            f"Resource {resource_id} was not found"
        )
```

Custom exceptions become especially valuable when your programs move beyond small scripts and start containing **business rules, multiple modules, APIs, databases, and different layers of application logic**.

Visit haas.dev for more resources and guides.

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

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