# **Errors & Exceptions**

There are two major kinds of errors:

1. Syntax Errors
2. Exceptions

## Syntax Errors

Syntax errors are parsing errors which occur when the code is not adhering to **Python Syntax**.

if True print("Hello")

SyntaxError: invalid synt

When there is a syntax error, the program will **not** execute even if that part of code is not used.

#### **Code**

print("Hello")

def greet():

print("World"

SyntaxError: unexpected EOF while

Notice that in the above code, the syntax error is inside the greet function, which is not used in rest of the code.

## Exceptions

Even when a statement or expression is **syntactically correct**, it may cause an **error** when an attempt is made to execute it.

Errors detected during execution are called **exceptions**.

### Example Scenario

We wrote a program to download a Video over the Internet.

* Internet is disconnected during the download
* We do not have space left on the device to download the video

*Example 1*

### Division Example

Input given by the user is not within expected values.

#### **Code**

def divide(a, b):

return a / b

divide(5, 0)

ZeroDivisionError: division b

*Example 2*

Input given by the user is not within expected values.

#### **Code**

def divide(a, b):

return a / b

divide("5", "10")

TypeError: unsupported operand type(s) for /: 'str

*Example 3*

Consider the following code, which is used to update the quantity of items in store.

#### **Code**

class Store:

def \_\_init\_\_(self):

self.items = {

"milk" : 20, "bread" : 30, }

def add\_item(self, name, quantity):

self.items[name] += quantity

s = Store()

s.add\_item('biscuits', 10)

#### **Output**

KeyError: 'bisc

## Working With Exceptions

What happens when your code runs into an exception during execution?

**The application/program crashes.**

### End-User Applications

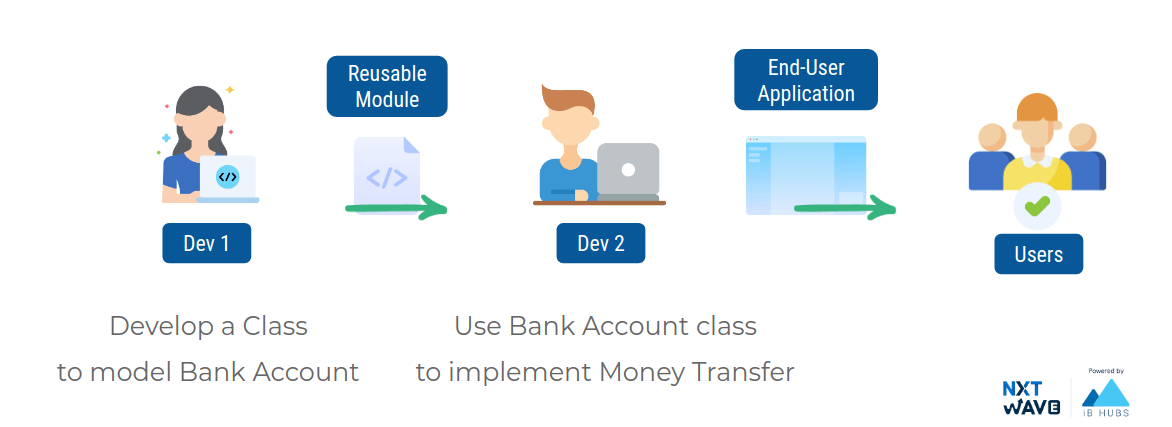
When you develop applications that are directly used by end-users, you need to **handle different possible exceptions** in your code so that the application will not crash.

### Reusable Modules

When you develop modules that are used by other developers, you should **raise exceptions** for different scenarios so that other developers can handle them.

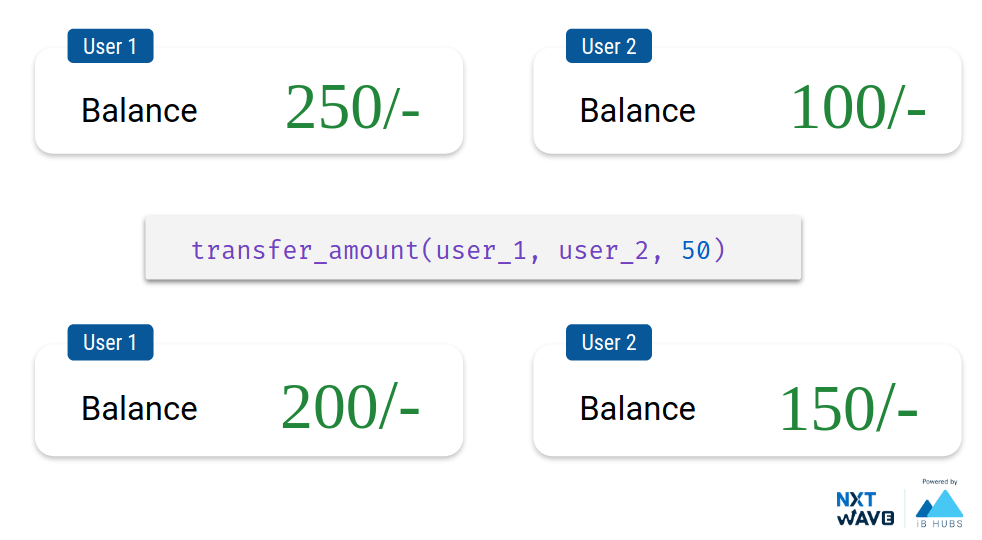
## Money Transfer App Scenario

Let’s consider we are creating an app that allows users to transfer money between them.



### Bank Account Class

*Example 1*

**

#### **Code**

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

else:

print("Insufficient Funds")

def deposit(self, amount):

self.balance += amount

def transfer\_amount(acc\_1, acc\_2, amount):

acc\_1.withdraw(amount)

acc\_2.deposit(amount)

user\_1 = BankAccount("001")

user\_2 = BankAccount("002")

user\_1.deposit(250)

user\_2.deposit(100)

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

transfer\_amount(user\_1, user\_2, 50)

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

#### **Output**

User 1 Balance: 250/-

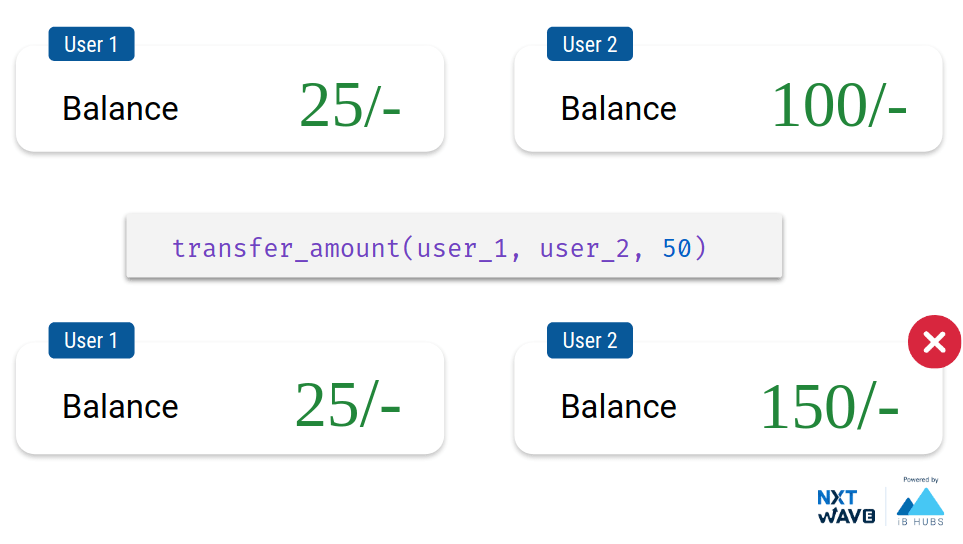
User 2 Balance: 100/-

Transferring 50/- from User 1 to User 2

User 1 Balance: 200/-

User 2 Balance: 15

*Example 2*

**

#### **Code**

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

else:

print("Insufficient Funds")

def deposit(self, amount):

self.balance += amount

def transfer\_amount(acc\_1, acc\_2, amount):

acc\_1.withdraw(amount)

acc\_2.deposit(amount)

user\_1 = BankAccount("001")

user\_2 = BankAccount("002")

user\_1.deposit(25)

user\_2.deposit(100)

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

transfer\_amount(user\_1, user\_2, 50)

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

#### **Output**

User 1 Balance: 25/-

User 2 Balance: 100/-

Insufficient Funds

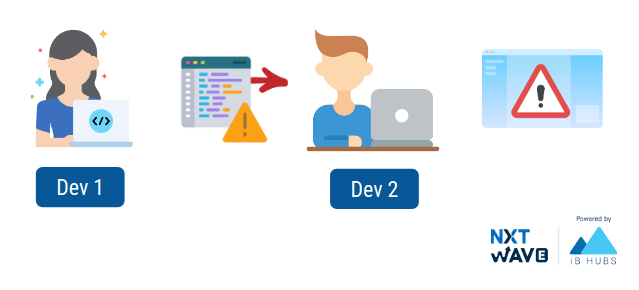
Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

User 2 Balance:

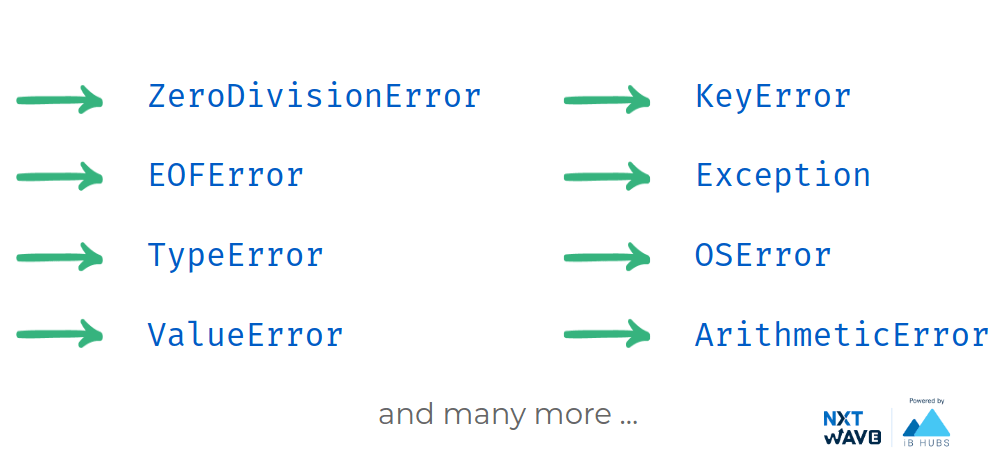
## Raising Exceptions

When your code enters an unexpected state, **raise** an exception to communicate it.



## Built-in Exceptions

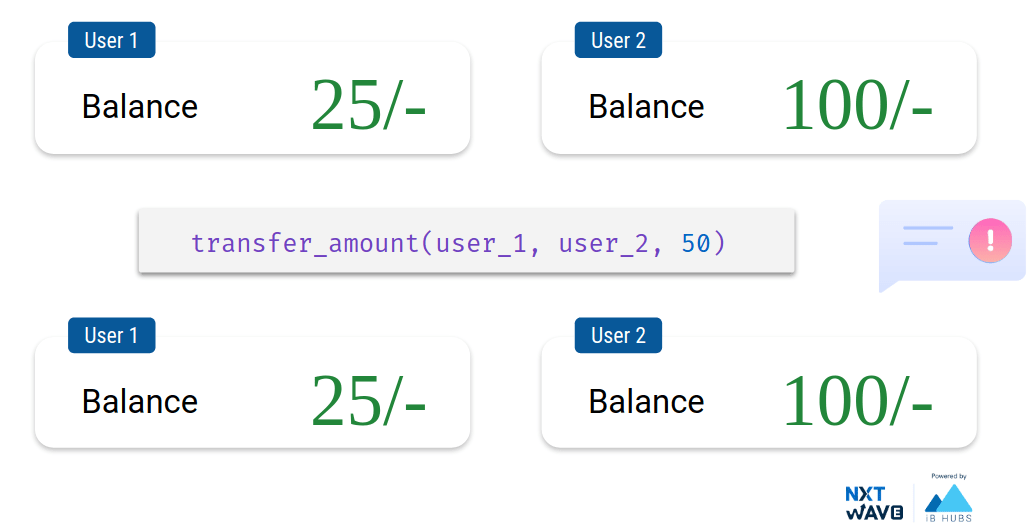
Different **exception classes** which are raised in different scenarios.



You can use the built-in exception classes with **raise** keyword to **raise an exception** in the program.

### Transfer Amount

*Example 1*

**

#### **Code**

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

else:

raise ValueError("Insufficient Funds")

def deposit(self, amount):

self.balance += amount

def transfer\_amount(acc\_1, acc\_2, amount):

try:

acc\_1.withdraw(amount)

acc\_2.deposit(amount)

return True

except:

return False

user\_1 = BankAccount("001")

user\_2 = BankAccount("002")

user\_1.deposit(25)

user\_2.deposit(100)

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

print(transfer\_amount(user\_1, user\_2, 50))

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

#### **Output**

User 1 Balance: 25/-

User 2 Balance: 100/-

False

Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

User 2 Balance: 100/-

### Summary

**Reusable Modules**

* While developing reusable modules, we need to raise Exceptions to stop our code from being used in a bad way.

**End-User Applications**

* While developing end-user applications, we need to handle Exceptions so that application will not crash when used.

## Handling Specific Exceptions

We can specifically mention the **name of exception** to catch all exceptions of that specific type.

#### **Syntax**

try:

# Write code that

# might cause exceptions.

except Exception:

# The code to be run when

# there is an exception.

#### **Code**

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except:

print("Unhandled Exception")

code:

5

0

#### **Output**

Denominator can't

#### **Code**

Input given by the user is not within expected values.

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except:

print("Unhandled Exception")

#### **Input**

12

a

#### **Output**

#### Unhandled Excep

We can also access the handled exception in an **object**.

#### **Syntax**

try:

# Write code that

# might cause exceptions.

except Exception as e:

# The code to be run when

# there is an exception.

#### **Code**

class BankAccount:

def \_\_init\_\_(self, account\_number):

self.account\_number = str(account\_number)

self.balance = 0

def get\_balance(self):

return self.balance

def withdraw(self, amount):

if self.balance >= amount:

self.balance -= amount

else:

raise ValueError("Insufficient Funds")

def deposit(self, amount):

self.balance += amount

def transfer\_amount(acc\_1, acc\_2, amount):

try:

acc\_1.withdraw(amount)

acc\_2.deposit(amount)

return True

except ValueError as e:

print(str(e))

print(type(e))

print(e.args)

return False

user\_1 = BankAccount("001")

user\_2 = BankAccount("002")

user\_1.deposit(25)

user\_2.deposit(100)

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

print(transfer\_amount(user\_1, user\_2, 50))

print("Transferring 50/- from User 1 to User 2")

print("User 1 Balance: {}/-".format(user\_1.get\_balance()))

print("User 2 Balance: {}/-".format(user\_2.get\_balance()))

#### **Output**

User 1 Balance: 25/-

User 2 Balance: 100/-

Insufficient Funds

<class 'ValueError'>

('Insufficient Funds',)

False

Transferring 50/- from User 1 to User 2

User 1 Balance: 25/-

User 2 Balance: 100/-

## Handling Multiple Exceptions

We can write **multiple exception blocks** to handle different types of exceptions differently.

#### **Syntax**

try:

# Write code that

# might cause exceptions.

except Exception1:

# The code to be run when

# there is an exception.

except Exception2:

# The code to be run when

# there is an exception.

#### **Code**

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except ValueError:

print("Input should be an integer")

except:

print("Something went wrong")

#### **Input**

5

0

#### **Output**

Denominator can't

#### **Code**

try:

a = int(input())

b = int(input())

c = a/b

print(c)

except ZeroDivisionError:

print("Denominator can't be 0")

except ValueError:

print("Input should be an integer")

except:

print("Something went wrong")

#### **Input**

12

a

#### **Output**

**Input should be an integer**