Error and Exception Handling

- Python may stop execution of a program because of two reasons.
    - **Syntax Errors**
        - An **error** is an action that is incorrect or inaccurate.
            - This is caused by wrong syntax in the code. It leads to the termination of the program.
    - **Exceptions**
        - Exceptions are raised when some internal events occur which change the normal flow of the program.
        - Some of the exceptions are listed below.

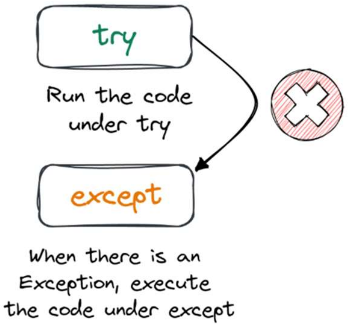
Exception	Description
`ArithmeticError`	Raised when an error occurs in numeric calculations
`AssertionError`	Raised when an assert statement fails
`AttributeError`	Raised when attribute reference or assignment fails
`EOFError`	Raised when the `input()` method hits an "end of file" condition (EOF)
`FloatingPointError`	Raised when a floating point calculation fails
`GeneratorExit`	Raised when a generator is closed (with the `close()` method)
`ImportError`	Raised when an imported module does not exist
`IndentationError`	Raised when indentation is not correct
`IndexError`	Raised when an index of a sequence does not exist
`KeyError`	Raised when a key does not exist in a dictionary
`KeyboardInterrupt`	Raised when the user presses Ctrl+c, Ctrl+z or Delete
`LookupError`	Raised when errors raised cant be found
`MemoryError`	Raised when a program runs out of memory
`NameError`	Raised when a variable does not exist
`NotImplementedError`	Raised when an abstract method requires an inherited class to override the method
`OSError`	Raised when a system related operation causes an error
`OverflowError`	Raised when the result of a numeric calculation is too large
`ReferenceError`	Raised when a weak reference object does not exist
`RuntimeError`	Raised when an error occurs that do not belong to any specific exceptions
`StopIteration`	Raised when the `next()` method of an iterator has no further values
`SyntaxError`	Raised when a syntax error occurs
`TabError`	Raised when indentation consists of tabs or spaces
`SystemError`	Raised when a system error occurs
`SystemExit`	Raised when the `sys.exit()` function is called
`TypeError`	Raised when two different types are combined
`UnboundLocalError`	Raised when a local variable is referenced before assignment
`UnicodeError`	Raised when a unicode problem occurs
`UnicodeEncodeError`	Raised when a unicode encoding problem occurs
`UnicodeDecodeError`	Raised when a unicode decoding problem occurs
`UnicodeTranslateError`	Raised when a unicode translation problem occurs
`ValueError`	Raised when there is a wrong value in a specified data type
`ZeroDivisionError`	Raised when the second operator in a division is zero

- It’s important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing.

Why use Exception

- **Standardized error handling**: Using built-in exceptions or creating a custom exception with a more precise name and description, you can adequately define the error event, which helps you debug the error event.
- **Cleaner code**: Exceptions separate the error-handling code from regular code, which helps us to maintain large code easily.
- **Robust application**: With the help of exceptions, we can develop a solid application, which can handle error event efficiently
- **Exceptions propagation**: By default, the exception propagates the call stack if you don’t catch it. For example, if any error event occurred in a nested function, you do not have to explicitly catch-and-forward it; automatically, it gets forwarded to the calling function where you can handle it.
- **Different error types**: Either you can use built-in exception or create your custom exception and group them by their generalized parent class, or Differentiate errors by their actual class

Exception Handling using try ….. catch

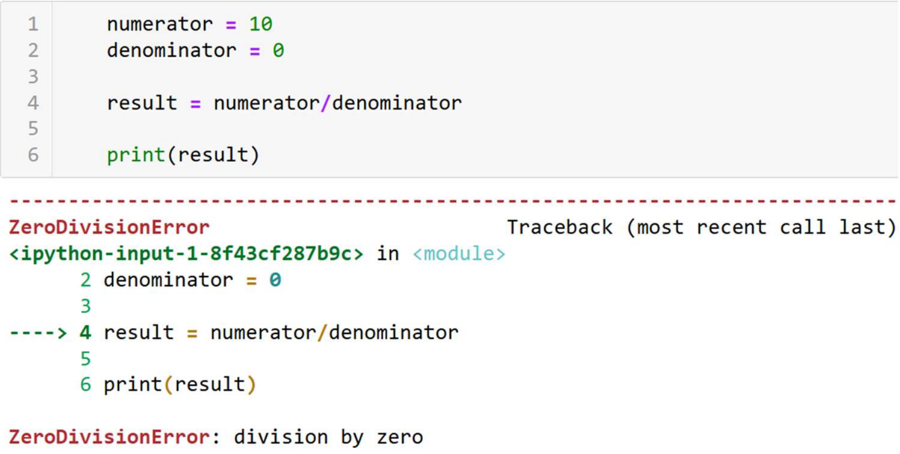


- Since exceptions abnormally terminate the execution of a program, it is important to handle exceptions. In Python, we use the `try...except` block to handle exceptions.

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

- Here, we have placed the code that might generate an exception inside the `try` block. Every try block is followed by an `except` block.
- When an exception occurs, it is caught by the `except` block. The `except` block cannot be used without the `try` block.

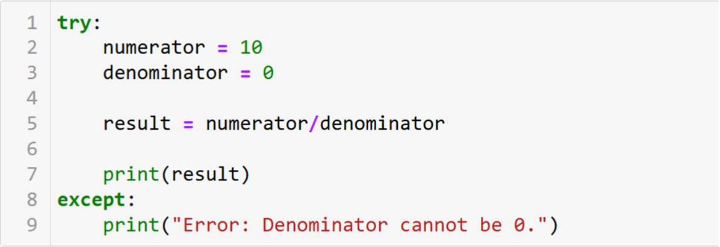
Example: Without Exception Handling



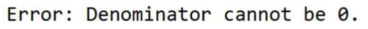
```
numerator = 10
denominator = 0

result = numerator/denominator

print(result)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-8f43cf287b9c> in <module>
      2 denominator = 0
      3
----> 4 result = numerator/denominator
      5
      6 print(result)

ZeroDivisionError: division by zero
```

Example: Exception Handling using try.....catch



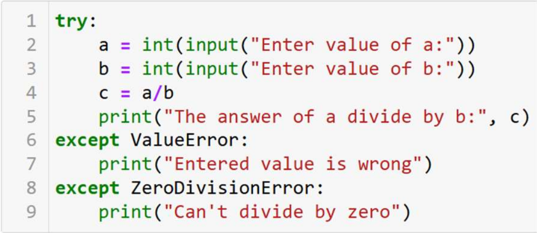
```python
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")
```



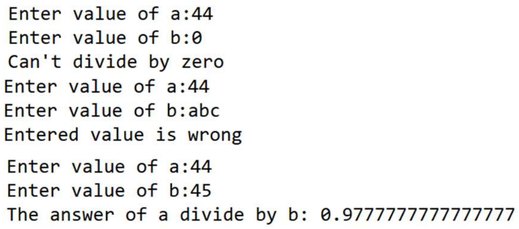
```
Error: Denominator cannot be 0.
```

- For each `try` block, there can be zero or more `except` blocks. Multiple `except` blocks allow us to handle each exception differently.
- The argument type of each `except` block indicates the type of exception that can be handled by it.

Example: Handling multiple exceptions using try.....catch



```python
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a/b
    print("The answer of a divide by b:", c)
except ValueError:
    print("Entered value is wrong")
except ZeroDivisionError:
    print("Can't divide by zero")
```

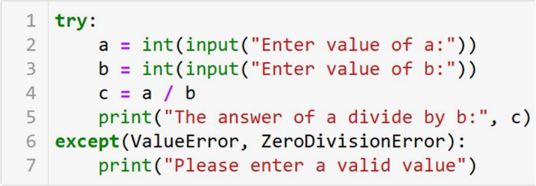


```
Enter value of a:44
Enter value of b:0
Can't divide by zero
Enter value of a:44
Enter value of b:abc
Entered value is wrong

Enter value of a:44
Enter value of b:45
The answer of a divide by b: 0.9777777777777777
```

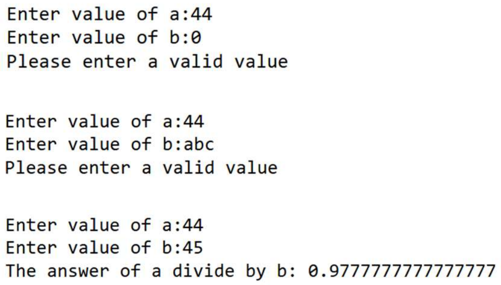
Handle multiple exceptions with a single except clause

- We can also handle multiple exceptions with a single except clause.
- For that, we can use a `tuple` of values to specify multiple exceptions in an `except` clause.

Example: Handling multiple exceptions using a single except



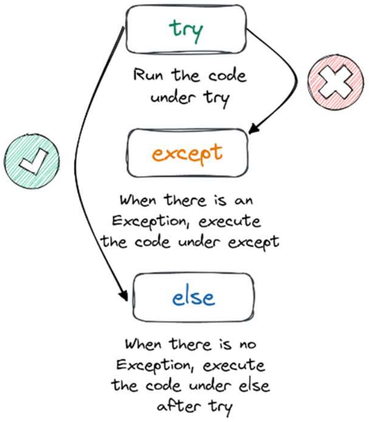
```python
try:
    a = int(input("Enter value of a:"))
    b = int(input("Enter value of b:"))
    c = a / b
    print("The answer of a divide by b:", c)
except(ValueError, ZeroDivisionError):
    print("Please enter a valid value")
```



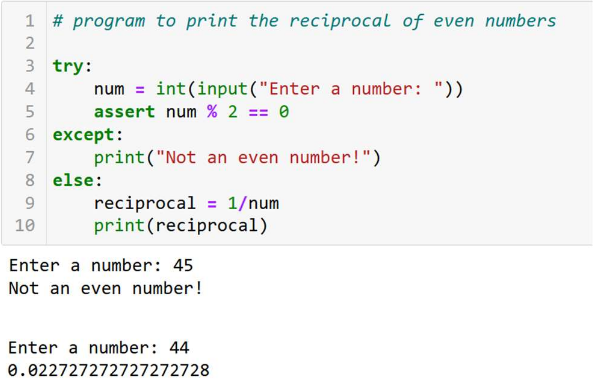
```
Enter value of a:44
Enter value of b:0
Please enter a valid value


Enter value of a:44
Enter value of b:abc
Please enter a valid value


Enter value of a:44
Enter value of b:45
The answer of a divide by b: 0.9777777777777777
```

Exception handling with try.....catch.....else

- In some situations, we might want to run a certain block of code if the code block inside `try` runs without any errors.
- For these cases, you can use the optional `else` keyword with the `try` statement.


try
Run the code under try
except
When there is an Exception, execute the code under except
else
When there is no Exception, execute the code under else after try

Example: Handling multiple exceptions using try.....catch.....else

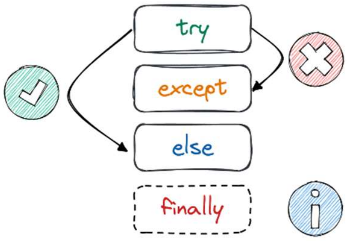


```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

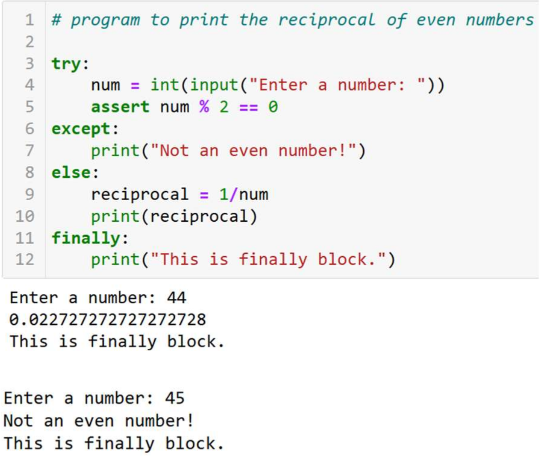
```
Enter a number: 45
Not an even number!


Enter a number: 44
0.022727272727272728
```

Finally Keyword in Python

- The `finally` keyword in the `try-except` block is always executed, irrespective of whether there is an exception or not.
- In simple words, the `finally` block of code is run after the `try`, except, the `else` block is final. It is quite useful in cleaning up resources and closing the object, especially closing the files.
- The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.



Example: Handling multiple exceptions using try.....catch.....else.....finally



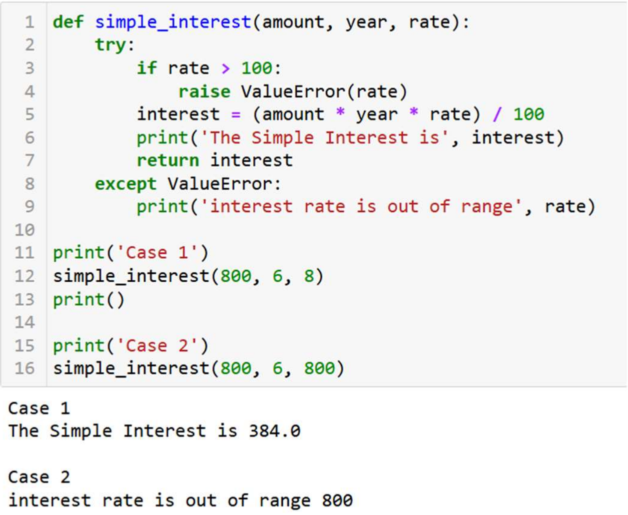
```
# program to print the reciprocal of even numbers

try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
finally:
    print("This is finally block.")
```

```
Enter a number: 44
0.022727272727272728
This is finally block.


Enter a number: 45
Not an even number!
This is finally block.
```

Raising an Exceptions

- In Python, the `raise` statement allows us to throw an exception. The single arguments in the `raise` statement show an exception to be raised. This can be either an exception object or an Exception class that is derived from the Exception class.
- The `raise` statement is useful in situations where we need to raise an exception to the caller program. We can raise exceptions in cases such as wrong data received or any validation failure.
- Follow the below steps to raise an exception:
  - Create an exception of the appropriate type. Use the existing built-in exceptions or create your own exception as per the requirement.
  - Pass the appropriate data while raising an exception.
  - Execute a raise statement, by providing the exception class.

Example: Raising an exception



```python
def simple_interest(amount, year, rate):
    try:
        if rate > 100:
            raise ValueError(rate)
        interest = (amount * year * rate) / 100
        print('The Simple Interest is', interest)
        return interest
    except ValueError:
        print('interest rate is out of range', rate)

print('Case 1')
simple_interest(800, 6, 8)
print()

print('Case 2')
simple_interest(800, 6, 800)
```

```
Case 1
The Simple Interest is 384.0

Case 2
interest rate is out of range 800
```