Python Exceptions

An exception is an unexpected event that occurs during program execution. For example,

```
divide_by_zero = 7 / 0
```

The above code causes an exception as it is not possible to divide a number by 0.

Let's learn about Python Exceptions in detail.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we

- try to open a file(for reading) that does not exist (`FileNotFoundError`)
- try to divide a number by zero (`ZeroDivisionError`)
- try to import a module that does not exist (`ImportError`) and so on.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

```
divide_numbers = 7 / 0

prit(divide_numbers)
```

Output

```
Traceback (most recent call last):
 File "<string>", line 1, in <module>
ZeroDivisionError: division by zero
```

Python Exceptions

An exception is an unexpected event that occurs during program execution. For example,

```
divide_by_zero = 7 / 0
```

The above code causes an exception as it is not possible to divide a number by 0.

Let's learn about Python Exceptions in detail.

Python Logical Errors (Exceptions)

Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.

For instance, they occur when we

- **try to open a file(for reading) that does not exist (`FileNotFoundError`)**
- **try to divide a number by zero (`ZeroDivisionError`)**
- **try to import a module that does not exist (`ImportError`) and so on.**

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Let's look at how Python treats these errors:

```
divide_numbers = 7 / 0

prit(divide_numbers)
```

Output

```
Traceback (most recent call last):

 File "<string>", line 1, in <module>
```

```
ZeroDivisionError: division by zero
```

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

1. **statement 1;**
2. **statement 2;**
3. **statement 3;**
4. **statement 4;**
5. **statement 5;//exception occurs**
6. **statement 6;**
7. **statement 7;**
8. **statement 8;**
9. **statement 9;**
10. **statement 10;**

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Python

Python Exception Handling

In the last tutorial, we learned about Python exceptions. We know that exceptions abnormally terminate the execution of a program.

This is why it is important to handle exceptions. In Python, we use the `try...except` block

---

Python try...except Block

The `try...except` block is used to handle exceptions in Python. Here's the syntax of `try...except` block:

```
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: Exception Handling Using try...except

```
try:
```

```
    numerator = 10
    denominator = 0

    result = numerator/denominator

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

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

In the example, we are trying to divide a number by 0. Here, this code generates an exception.

To handle the exception, we have put the code, `result = numerator/denominator` inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped.

The `except` block catches the exception and statements inside the `except` block are executed.

If none of the statements in the `try` block generates an exception, the `except` block is skipped.

Catching Specific Exceptions in Python

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. For example,

```
try:

    even_numbers = [2,4,6,8]
```

```
    print(even_numbers[5])

except ZeroDivisionError:
    print("Denominator cannot be 0.")

except IndexError:
    print("Index Out of Bound.")

# Output: Index Out of Bound
```

In this example, we have created a list named `even_numbers`.

Since the list index starts from 0, the last element of the list is at index 3. Notice the statement,

```
print(even_numbers[5])
```

Here, we are trying to access a value to the index 5. Hence, `IndexError` exception occurs.

When the `IndexError` exception occurs in the `try` block,

- The `ZeroDivisionError` exception is skipped.
- The set of code inside the `IndexError` exception is executed.

Python try with else clause

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.

Let's look at an example:

```
# 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)
```

Output

If we pass an odd number:

```
Enter a number: 1
Not an even number!
```

If we pass an even number, the reciprocal is computed and displayed.

```
Enter a number: 4
0.25
```

However, if we pass 0, we get `ZeroDivisionError` as the code block inside `else` is not handled by preceding `except`.

```
Enter a number: 0
Traceback (most recent call last):
  File "<string>", line 7, in <module>
    reciprocal = 1/num
ZeroDivisionError: division by zero
```

> Note: Exceptions in the `else` clause are not handled by the preceding except clauses.

Python try...finally

In Python, the `finally` block is always executed no matter whether there is an exception or not.

The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

Let's see an example,

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

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

finally:
    print("This is finally block.")
```

Output

```
Error: Denominator cannot be 0.
This is finally block.
```

In the above example, we are dividing a number by 0 inside the `try` block. Here, this code generates an exception.

The exception is caught by the `except` block. And, then the `finally` block is executed.

Python Custom Exceptions

In the previous tutorial, we learned about different built-in exceptions in Python and why it is important to handle exceptions. .

However, sometimes we may need to create our own custom exceptions that serve our purpose.

Defining Custom Exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in `Exception` class.

Here's the syntax to define custom exceptions,

```
class CustomError(Exception):
    ...
    pass

try:
   ...

except CustomError:
    ...
```

Here, `CustomError` is a user-defined error which inherits from the `Exception` class.

Note:

- When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file.
- Many standard modules define their exceptions separately as `exceptions.py` or `errors.py` (generally but not always).

Example: Python User-Defined Exception

```
# define Python user-defined exceptions
class InvalidAgeException(Exception):
    "Raised when the input value is less than 18"
    pass

# you need to guess this number
number = 18

try:
    input_num = int(input("Enter a number: "))
    if input_num < number:
        raise InvalidAgeException
    else:
        print("Eligible to Vote")

except InvalidAgeException:
    print("Exception occurred: Invalid Age")
```

Output

If the user input `input_num` is greater than 18,

```
Enter a number: 45
Eligible to Vote
```

If the user input `input_num` is smaller than 18,

```
Enter a number: 14
Exception occurred: Invalid Age
```

In the above example, we have defined the custom exception `InvalidAgeException` by creating a new class that is derived from the built-in `Exception` class.

Here, when `input_num` is smaller than 18, this code generates an exception.

When an exception occurs, the rest of the code inside the `try` block is skipped.

The `except` block catches the user-defined `InvalidAgeException` exception and statements inside the `except` block are executed.

Customizing Exception Classes

We can further customize this class to accept other arguments as per our needs.

To learn about customizing the Exception classes, you need to have the basic knowledge of Object-Oriented programming.

Visit Python Object Oriented Programming to learn about Object-Oriented programming in Python.

Let's see an example,

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary, message="Salary is not in (5000, 15000)
range"):
        self.salary = salary
        self.message = message
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

Output

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "<string>", line 17, in <module>
    raise SalaryNotInRangeError(salary)
__main__.SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Here, we have overridden the constructor of the `Exception` class to accept our own custom arguments `salary` and `message`.

Then, the constructor of the parent `Exception` class is called manually with the `self.message` argument using `super()`.

The custom `self.salary` attribute is defined to be used later.

The inherited `__str__` method of the `Exception` class is then used to display the corresponding message when `SalaryNotInRangeError` is raised.