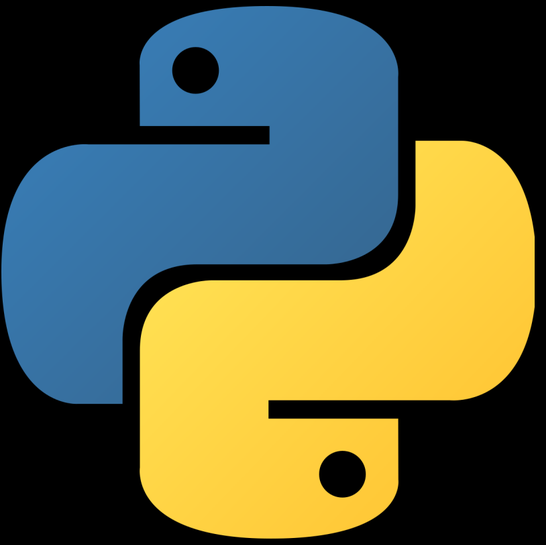


Prof R Madana Mohana

OBJECT ORIENTED PROGRAMMING USING PYTHON

ERROR & EXCEPTION HANDLING

Raising Exceptions | Built-in and User-defined Exceptions

https://www.youtube.com/c/RASINENIMADANAMOHANA

OUTLINE

- Raising Exceptions
- Instantiating Exceptions
- Handling Exceptions in Invoked Functions
- Built-in and User-defined Exceptions

RAISING EXCEPTIONS

- We can deliberately **raise an exception** using the **raise** keyword.

The general syntax for the **raise** statement is:

```
raise [Exception [, args [, traceback]]]
```

RAISING EXCEPTIONS

```
raise [Exception [, args [, traceback]]]
```

- Here, **Exception** is the name of exception to be raised (example, **TypeError**).
- **args** is optional and specifies a value for the exception argument.
- If **args** is not specified, then the exception argument is None.
- The final argument, ***traceback***, is also optional and if present, is the ***traceback*** object used for the exception .

RAISING EXCEPTIONS

Ex1. Program to deliberately raise an exception

```
try:
    num = 10
    print(num)
    raise ValueError
except:
    print("Exception occurred...Program Terminating..")
```

OUTPUT:

```
10
Exception occurred...Program Terminating..
```

RAISING EXCEPTIONS

Ex2. Program to re-raise an exception

```
try:
    raise NameError
except:
    print("Re-Raising the Exception..")
    raise
```

OUTPUT:

```
Re-Raising the Exception..
Traceback (most recent call last):
  File "F:/RMM OOPP UNIT-5/Exceptions-Re-Raise.py", line 3, in
<module>
    raise NameError
NameError
```

INSTANTIATING EXCEPTIONS

- **Python** allows programmers to **instantiate an exception** first before raising it and add any attributes (or arguments) to it as desired.
- These attributes can be used to give additional information about the error.
- To i**nstantiate the exception**, the **except block** may specify a variable after the exception name.

INSTANTIATING EXCEPTIONS

- The variable then becomes an **exception instance** with the arguments stored in **instance.args**.
- The **exception instance** also has the **__str__()** method defined so that the arguments can be printed directly without using **instance.args**.
- The contents of the argument vary based on exception type.

INSTANTIATING EXCEPTIONS

Ex1. Program to understand the process of instantiating an exception.

```
try:
    raise Exception("Hello", "World")
except Exception as errorObj:
    print(type(errorObj)) # the exception instance
    print(errorObj.args) # arguments stored in .args
    print(errorObj) # __str__() allows args to be printed
directly
    arg1, arg2 = errorObj.args
    print("Argument1 = ", arg1)
    print("Argument2 = ", arg2)
```

INSTANTIATING EXCEPTIONS

Ex1. Program to understand the process of instantiating an exception.

OUTPUT:

```
<class 'Exception'>
('Hello', 'World')
('Hello', 'World')
Argument1 =  Hello
Argument2 =  World
```

Note:

If we raise an exception with arguments but do not handle it, then the name of the exception is printed along with its arguments.

INSTANTIATING EXCEPTIONS

Ex2. Program to raise an exception with arguments

```
try:
    raise Exception("Hello", "World")
except ValueError:
    print("Program Terminating..")
```

OUTPUT:

```
Traceback (most recent call last):
  File "F:/RMM OOPP UNIT-5/Exceptions-rasie with
arguments.py", line 3, in <module>
    raise Exception("Hello", "World")
Exception: ('Hello', 'World')
```

HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

Exceptions can also be handled inside functions that are called in the **try block** as shown in the program given below:

Ex1.Program to handle exceptions from an invoked function

```
def Divide(num, denom):
    try:
        quo = num/denom
        print("Quotient = :",quo)
    except ZeroDivisionError:
        print("You cannot divide a number by zero..Program
terminating..")
Divide(20,0)
Divide(20,4)
```

HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

Exceptions can also be handled inside functions that are called in the **try block** as shown in the program given below:

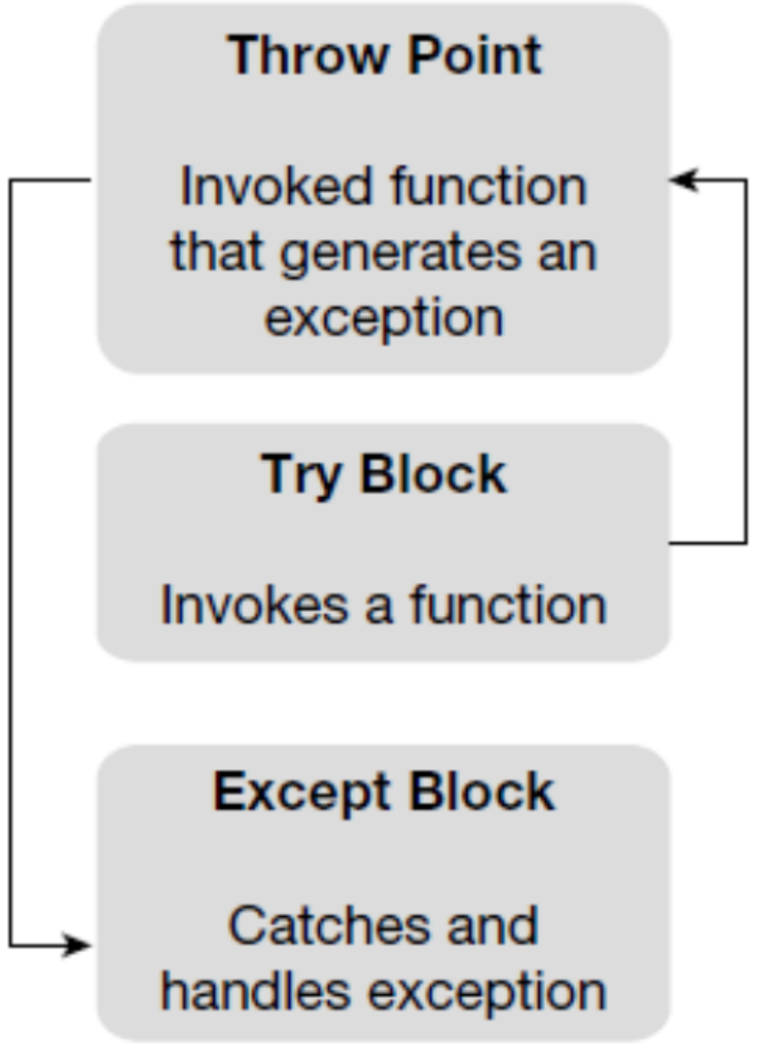
Ex1.Program to handle exceptions from an invoked function

OUTPUT:

```
You cannot divide a number by zero..Program terminating..
Quotient = : 5.0
```

HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

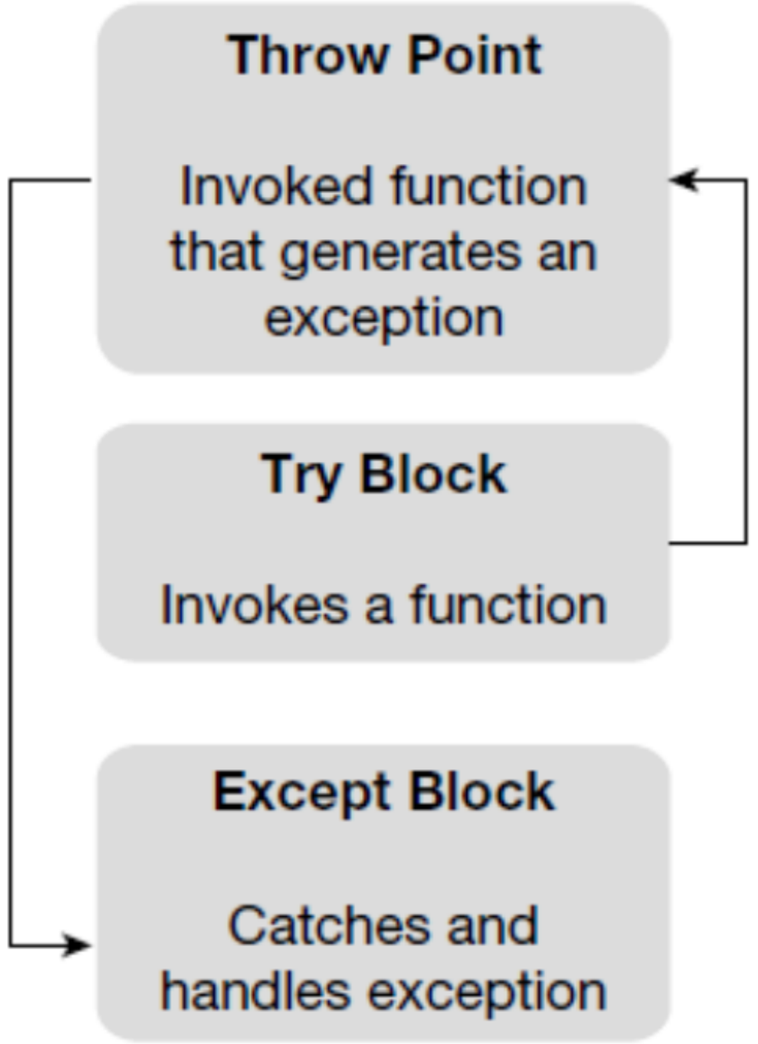
- Basically, a large program is usually divided into **n** number of functions.
- The possibility that the **invoked function** may generate an exceptional condition cannot be ignored.
- The figure, shows the scenario when the function invoked by the **try block** throws an exception which is handled by the **except block** in the calling function.



HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

The syntax of such situation can be given as:

```
function_name(arg list):
    -------
    -------
try:
    ----
    function_name() //Function call
    ----
except ExceptionName:
    ------
    # Code to handle exception
    ------
```



HANDLING EXCEPTIONS IN INVOKED FUNCTIONS

Ex.Program to handle exceptions in the calling function

```
def Divide(num, denom):
    try:
        quo = num/denom
        print("Quotient = :",quo)
    except ZeroDivisionError:
        print("You cannot divide a number by zero..Program
terminating..")
Divide(20,0)
Divide(20,4)
```

OUTPUT:

```
Quotient = 2.857142857142857
You cannot divide a number by zero..Program Terminating..
```

BUILT-IN AND USER-DEFINED EXCEPTIONS

Exceptions are categorized into two types:

1. Built-in Exceptions
2. User-defined Exceptions

Built-in Exceptions:

The **following table** lists some **standard exceptions** that are already defined in **Python**. These **built-in exceptions** force our program to output an error when something in it goes wrong.

BUILT-IN AND USER-DEFINED EXCEPTIONS

Built-in Exceptions:

Exception	Description
`Exception`	Base class for all exceptions
`StopIteration`	Generated when the `next()` method of an iterator does not point to any object
`SystemExit`	Raised by `sys.exit()` function
`StandardError`	Base class for all built-in exceptions (excluding `StopIteration` and `SystemExit`)
`ArithmeticError`	Base class for errors that are generated due to mathematical calculations
`OverflowError`	Raised when the maximum limit of a numeric type is exceeded during a calculation
`FloatingPointError`	Raised when a floating point calculation could not be performed
`ZeroDivisionError`	Raised when a number is divided by zero
`AssertionError`	Raised when the assert statement fails
`AttributeError`	Raised when attribute reference or assignment fails
`EOFError`	Raised when end-of-file is reached or there is no input for `input()` function
`ImportError`	Raised when an import statement fails
`KeyboardInterrupt`	Raised when the user interrupts program execution (by pressing Ctrl+C)
`LookupError`	Base class for all lookup errors
`IndexError`	Raised when an index is not found in a sequence
`KeyError`	Raised when a key is not found in the dictionary
`NameError`	Raised when an identifier is not found in local or global namespace (referencing a non-existent variable)
`UnboundLocalError`, `EnvironmentError`	Raised when an attempt is made to access a local variable in a function or method when no value has been assigned to it.

BUILT-IN AND USER-DEFINED EXCEPTIONS

Built-in Exceptions:

IOError	Raised when input or output operation fails (for example, opening a file that does not exist)
SyntaxError	Raised when there is a syntax error in the program
IndentationError	Raised when there is an indentation problem in the program
SystemError	Raised when an internal system error occurs
ValueError	Raised when the arguments passed to a function are of invalid data type or searching a list for a non-existent value
RuntimeError	Raised when the generated error does not fall into any of the above category
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not implemented
TypeError	Raised when two or more data types are mixed without coercion

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

- **Python** allows programmers to create their **own exceptions** by creating a new exception class.
- The new exception class is derived from the base class Exception which is pre-defined in **Python**.

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

Ex. Program to define a user-defined exception

```
class myError(Exception):
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return repr(self.val)
try:
    raise myError(30)
except myError as e:
    print("User-defined Exception Generated with value:
",e.val)
```

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

Ex. Program to define a user-defined exception

OUTPUT:

```
User-defined Exception Generated with value:  30
```

- In this program, the **__init__()** method of Exception class has been overridden by the new class.
- The customized exception class can be used to perform any task.
- However, these classes are usually kept simple and have only limited attributes to provide information about the error to be extracted by handlers for the exception.

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

- Creating our **own exception class** or defining a **user-defined exception** is known as ***custom exception***.

Note: An **exception** can be a string, a class, or an object. Most of the **exceptions** raised by **Python** are classes, with an argument that is an instance of the class.

- **as** is a keyword that allow programmers to name a variable within an **except statement**.

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

- When creating a **module** that can raise **different exceptions**, a better approach would be to create a **base class** for **exceptions** defined by that **module**, and **subclasses** to create specific exception classes for different error conditions.

User-defined Exceptions:

Ex2. Program to create sub-classes for Exception class to handle exceptions in a better customized way

```
class Error(Exception):
    def message(self):
        raise NotImplementedError()
class InputError(Error):
    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg
    def message(self):
        print("Error in input in Expression..")
        print(self.expr)
```

User-defined Exceptions:

Ex2. Program to create sub-classes for Exception class to handle exceptions in a better customized way

```
try:
    a = input("Enter the value of a:")
    raise InputError("input(\"Enter the value of a:s\")","Input Error")
except InputError as ie:
    ie.message()
```

OUTPUT:

```
Enter the value of a:20
Error in input in Expression..
input("Enter the value of a:s")
```

BUILT-IN AND USER-DEFINED EXCEPTIONS

User-defined Exceptions:

- Although there is no **naming convention** for naming a **user-defined exception**, it is better to define **exceptions** with names that end in "**Error**", to make it consistent with the naming of the standard exceptions.

Note: Many standard modules define their **own exceptions** to report **errors** that may occur in functions they define.