



Errors & Exceptions

- The program will stop
- Data will be created called an exception
- These activities are often referred to as raising an exception.
- When PYTHON has no idea what do to with your code it raises an exception
- If we do not take care of the raised exception the program will be terminated, and you will see an error message displayed in your console by Python
- If we handle the exception, the program can be resumed and its execution will continue.
- Python offers effective tools that allow us to observe exceptions, identify them and handle them
- Each potential exception can be identified by its name, so we can easily categorize and react accordingly

```
data = 1
```

```
data /= 0
```

**Traceback (most recent call last): File "div.py", line 2, in data /= 0
ZeroDivisionError: division by zero**

This specific exception error is called **ZeroDivisionError.**

```
list = []
```

```
x = list[0]
```

Traceback (most recent call last): File "lst.py", line 2, in x = list[0] IndexError: list index out of range

This is the **IndexError.**

Exceptions

- When you run an incorrect code, you will get error displayed on your console/screen
- For every type of error there is a specific exception to handle it

```
a = '7'  
try:  
    a += 7  
    print(a)  
except:  
    print('Cannot add a string to an integer')
```

Errors & Exceptions

try:

```
n = input("Enter a number: ")
```

```
y = 1 / n  
print(y)
```

except ZeroDivisionError:

```
print("You cannot divide by zero, sorry.")
```

except TypeError:

```
print('type error has occurred')
```

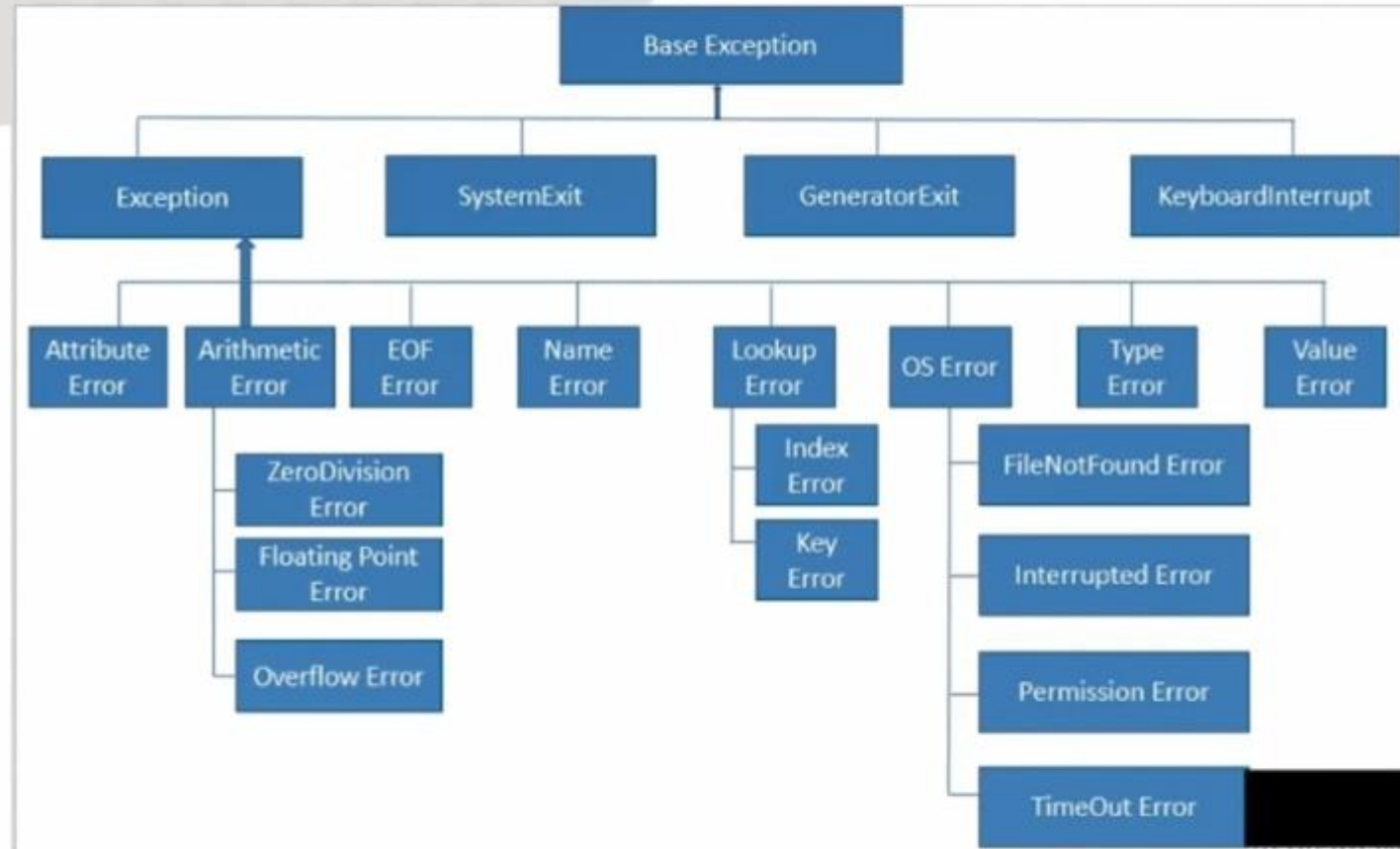
if you press Ctrl-C while the program is waiting for the user's input (an exception named KeyboardInterrupt will appear)

Errors & Exceptions

- There are 63 built-in exceptions
- Its tree-shaped hierarchy
- Exception located closer to the root can be considered more general (abstract)
- General exception can easily handle a raised exception as long as it matches the coverage, a concrete exception is not required

Example of an exception raised inside a function

```
def division(n):  
    return 1 / n  
try:  
    division(0)  
except ArithmeticError:  
    print('exception was raised!')
```



Errors & Exceptions

```
def division():  
    raise ZeroDivisionError
```

```
try:  
    division()  
except ArithmeticError:  
    print(" error!!")
```

The raise instruction raises the specified exception as if it was raised in a normal (natural) way:

Errors & Exceptions

- The raise instruction may also be utilized in the following way

```
def division(n):  
    try:  
        return n / 0  
    except:  
        print("error")  
        raise  
try:  
    division(1)  
except ArithmeticError:  
    print("not ok")
```

this type of raise instruction can only be used inside the except branch only, using it in any other context causes an error.

We can distribute the exception handling among different parts of the code by writing our scrip the following way.

The ZeroDivisionError is raised twice

Errors & Exceptions

```
import math
```

```
data = float(input("Enter a number: "))
```

```
assert data >= 2.0
```

```
x = math.sqrt(data)
```

```
print(x)
```

- Use assert to be absolutely safe from wrong data, and when you aren't absolutely sure that the data has been carefully examined

- Assertions is there to assist exceptions



Exceptions

```
def access(val):  
    try:  
        element = [1,2,3]  
        element[val]  
    except IndexError:  
        print("Index Error")  
        return val  
    else:  
        print("Success")  
        return val  
  
print(access(0))  
print(access(5))
```

Exceptions

```
def access(val):
```

```
    try:
```

```
        element = [1,2,3]  
        element[val]
```

```
    except IndexError:
```

```
        print("Index Error")  
        return val
```

```
    else:
```

```
        print("Success")  
        return val
```

```
    finally:
```

```
        print('always executed')
```

```
print(access(0))
```

```
print(access(5))
```

- finally block must always be the last branch of the try except block
- We don't have to place both else and finally within our try except block , it can be either one of them or both
- Most importantly, the finally block is always executed regardless of the result

Exceptions

- Exceptions are actual classes, when an exception is raised, an object of the class is instantiated [Object is being created], and the code goes through all levels of program execution, looking and searching for the adequate except branch that is prepared to deal with the specific exception.
- Exception as an object carries useful information
- In order for us to access an Exception object, we have to catch the object using the as keyword, then give the object an identifier, such as an alias

try:

```
a = [1,2,3]  
a[3]
```

except IndexError as Index:

```
print(Index)  
print(Index.__str__())
```

- By applying the `__str__` method on our Exception object will should receive a message in human readable form describing the cause behind the exception



Exceptions

```
def display(element):  
    length = len(element)  
    if length == 0:  
        print("Zero")  
    else:  
        print(element)  
  
try:  
    raise Exception("I am")  
except Exception as Ex:  
    print(Ex, Ex.__str__(), sep=' : ', end=' : ')  
    display(Ex.args)
```

I am : I am : ('I am',)

- The `BaseException` class has a property called `args`.
- `args` property is a tuple
- It gathers all arguments passed to the class constructor.
- It is empty of course if we don't pass any arguments to the constructor

Exceptions

```
class WarningEx(Exception):
    def __init__(self, name="", announcement=""):
        Exception.__init__(self, announcement)
        self.name = name

class ErrorExc(WarningEx):
    def __init__(self, name, announcement, date):
        WarningEx.__init__(self, name, announcement)
        self.date = date

def calla (name):
    if name == 'WarningEx':
        raise WarningEx('WarningEx' , 'This is a WARNING!!!!')
    if name == 'ErrorExc':
        raise ErrorExc('ErrorEx', "ERROR ERROR ERROR", 2024)

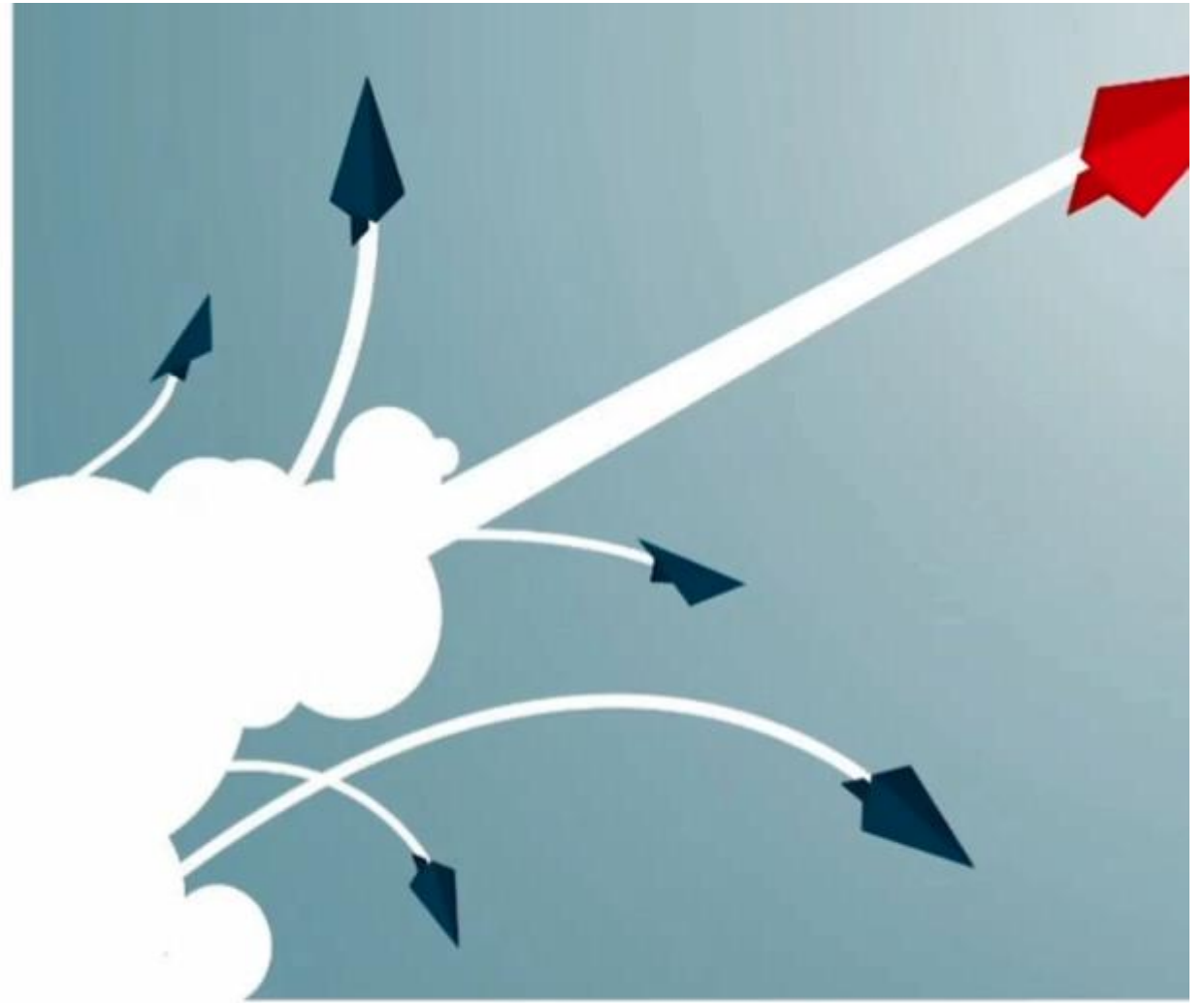
for x in ['WarningEx', 'ErrorExc']:
    try:
        calla(x)
    except ErrorExc as EX:
        print(EX, ': ', EX.date)
    except WarningEx as WE:
        print(WE, ': ', WE.name)
```

```
class PersonalIndexError(IndexError):
    pass
```

- If we want to create our own exception we can inherit from a specialized exception, however it will be closely connected to python's built-in exception tree
- If we want to inherit from our own hierarchy, and don't want it to be closely associated with Python's exception tree, we need to derive it from exception classes, like general Exception

Exceptions

```
try:  
    import file  
except ImportError as IE:  
    print(IE.name)  
    print(IE.path)
```



```
try:
    b'\x99'.decode("utf-8")
except UnicodeError as UE:
    print(UE.encoding)
    print(UE.reason)
    print(UE.object)
    print(UE.start)
    print(UE.end)
```

Chained Exceptions

- `__context__`
- `__cause__`

```
data = ['first', '2']
```

```
try:
```

```
    print(data[2])
```

```
except Exception as EX:
```

```
    print(1 / 0)
```

- During handling of the above exception, another exception occurred:



```
data = ['first', '2']
```

```
try:
```

```
    print(data[2])
```

```
except Exception as EX:
```

```
    try:
```

```
        print(1 / 0)
```

```
    except ZeroDivisionError as Z:
```

```
        print( EX)
```

```
        print(Z)
```

```
        print(Z.__context__)
```

```
class NewError(Exception):
    pass

def check():
    try:
        print(data[2])
    except IndexError as IN:
        raise NewError('New Error') from IN

data = ['first', '2']

try:
    check()
except NewError as NE:
    print('Exception: "{}", was raised due to "{}"'.format(NE, NE.__cause__))
```

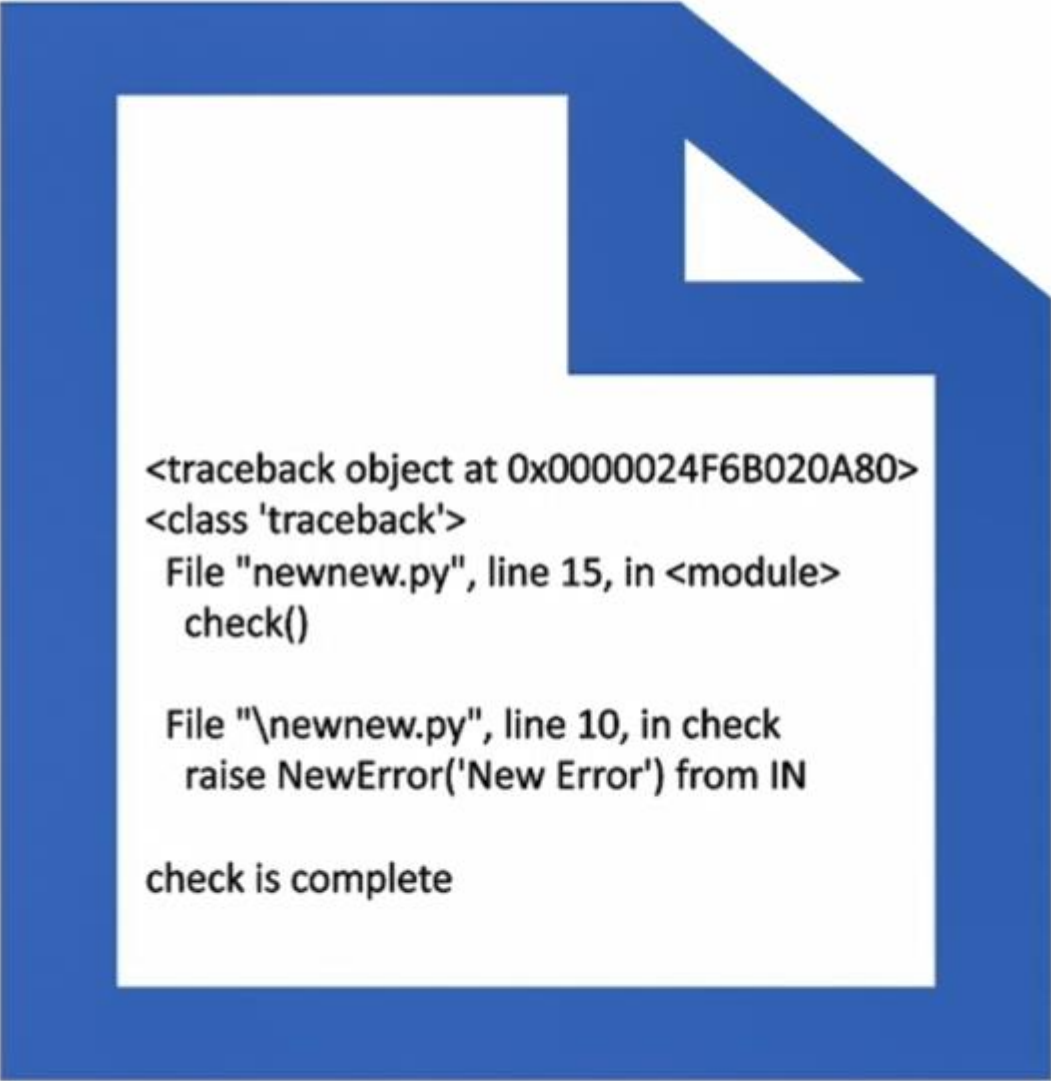



```
class NewError(Exception):
    pass

def check():
    try:
        print(data[2])
    except IndexError as IN:
        raise NewError('New Error') from IN

data = ['first', '2']

try:
    check()
except NewError as NE:
    print(NE.__traceback__)
    print(type(NE.__traceback__))
```



```
<traceback object at 0x0000024F6B020A80>  
<class 'traceback'>  
File "newnew.py", line 15, in <module>  
    check()  
  
File "\newnew.py", line 10, in check  
    raise NewError('New Error') from IN  
  
check is complete
```

```
import traceback  
  
class NewError(Exception):  
    pass  
  
def check():  
    try:  
        print(data[2])  
    except IndexError as IN:  
        raise NewError('New Error') from IN  
  
data = ['first', '2']  
  
try:  
    check()  
except NewError as NE:  
    print(NE.__traceback__)  
    print(type(NE.__traceback__))  
    more_info = traceback.format_tb(NE.__traceback__)  
    print('\n'.join(more_info))  
  
print('check is complete')
```