

**Exceptions**:

“*When writing programs, many things can go wrong, our programs may crash while encountering an error and terminate abruptly*”.

Usually these errors happen because of *programming mistakes* or bad data given by the user or *resources* *not* being *available*.

For example you might need to open a file but if that file does not exist your programs will crash. Instead we should *show a proper error message to the user* like, Hey! this file does not exist.

Take a look at this code for example,

numbers = [1, 2]

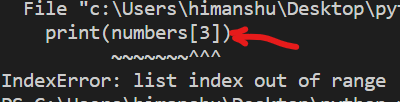
print(numbers[3])

I get an error in this case,

IndexError: list index out of range

In programming we refer to this type of error as an exception,

***“****An exception is a kind of error that terminates the execution of a program****”***.

 In programming we say this line,

*print(number[3])* threw an exception.

This exception was thrown by programmers mistake.

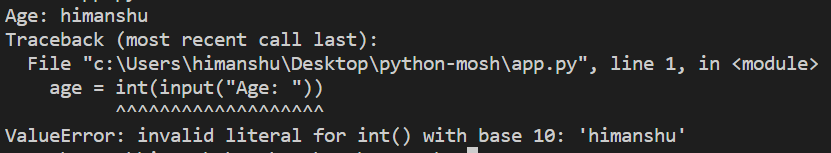
Consider another example,

age = int(input("Age: "))

We are asking user to enter their age, now we know that this *input* function returns a string, so we are first converting it into integer and then storing in *age*.

If we enter a non – numeric value like “himanshu”, our program crashes.

We get an of type, ValueError: invalid literal for int() with base 10: 'himanshu'



**Handling Exceptions**:

In this lecture we will learn how to handle exceptions in our program. Continuing from last example, we will handle ValueError exception.

First we need to put our statement in a try-except block,

try:

    age = int(input("Age: "))

except:

To handle the ValueError specifically, we add this in except statement and print a user friendly error message,

try:

    age = int(input("Age: "))

except ValueError:

    print("You did not enter a valid age")



***“****When python sees a* ***try*** *block it will execute every statement in this block, if any of those statements throw an exception, the code in the* ***except*** *clause will be executed****”***.

By adding except clause in our code, our program does not crash and the next line of code executes.

try:

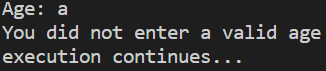
    age = int(input("Age: "))

except ValueError:

    print("You did not enter a valid age")

print("execution continues...")

Next line of code after try - except executes…



Note: We have an optional else block, which comes after except and if no exceptions were thrown is executed,

try:

    age = int(input("Age: "))

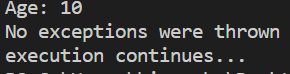
except ValueError:

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")

print("execution continues...")



So, what we have in else block will only be executed if no exceptions were thrown.

Note: we can also pass the exception as an object and display to the user.

try:

    age = int(input("Age: "))

except ValueError as ex:

    print("You did not enter a valid age")

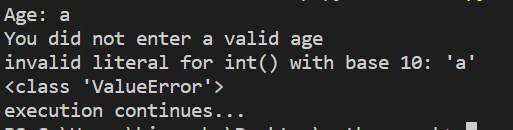
    print(ex)

    print(type(ex))

else:

    print("No exceptions were thrown")

print("execution continues...")



We can see the actual error message that is included in exception(*very technical to show to end user*) which is an object of type ValueError.

**Handling different exceptions**:

We will add a new exception in our previous example,

try:

    age = int(input("Age: "))

    xFactor = 10 / age

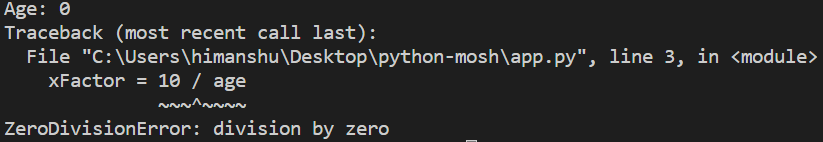
except ValueError:

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")

Now, When we entered age as 0, we get ZeroDivisonError.



Since we have not handled this particular exception, therefore our program crashes.

Now we add another exception block right below ValueError,

try:

    age = int(input("Age: "))

    xFactor = 10 / age

except ValueError:

    print("You did not enter a valid age")

except ZeroDivisionError:

    print("Age cannot be zero")

else:

    print("No exceptions were thrown")



There is a better way to handle exceptions. We can specify multiple exceptions in the same line like this,

try:

    age = int(input("Age: "))

    xFactor = 10 / age

except (ValueError, ZeroDivisionError):

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")



**Cleaning up**:

There are times when we need to work with external resources like files, network connections, databases and so on.

Whenever we use these resources, after we are done, we need to release them.

For example, when you open a file, we should always close it after we are done otherwise another process or another program may not be able to open that file.

In this example, let us open a file using *open* function which returns a file object and after we are done use *close* method of this object to close the file.

try:

    file = open("app.py")

    age = int(input("Age: "))

    xFactor = 10 / age

    file.close()

except (ValueError, ZeroDivisionError):

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")

Now the problem with this code is that this statement *file.close*() will not be executed in case of an exception.

In other words, if statements with *age* or *xFactor* throw an exception, control will be moved to except clause and *file.close*() will not be executed.

The only solution is to use the ***finally*** clause at the end, *This finally clause is always executed whether we have an exception or not and we use it to release external resources”*.

try:

    file = open("app.py")

    age = int(input("Age: "))

    xFactor = 10 / age

except (ValueError, ZeroDivisionError):

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")

finally:

    file.close()

**The With Statement**:

In the last lecture we learned that we used *finally* clause to release external resources. Now we have a shorter and cleaner way to achieve the same thing but without finally clause.

*But it does not always work, only with a certain type of objects*.

Instead of getting the return value from the *open* function we will prefix it with the ***with*** statement then we will get access to the return value of this function using ***as*** keyword.

try:

    with open("app.py") as file:

        print("file opened")

    age = int(input("Age: "))

    xFactor = 10 / age

except (ValueError, ZeroDivisionError):

    print("You did not enter a valid age")

else:

    print("No exceptions were thrown")

Inside the with code block, we can do any operation with this *file* object we get, we can read something from it or write into it.

Whenever we open a file using *with* statement, python will automatically call *file.close*() whether we have *finally* clause or not.

***“****In other words, the with statement is used to automatically release external resources****”***.

*How with works behind the scenes*?

Notice the file object and its members. Two methods(*magic methods*) associated with this file object are *\_\_enter\_\_* and *\_\_exit\_\_.*



Whenever an object has these two methods, we say that object supports **context management protocol** and that object can be used with the *with* statement.

Python will automatically call the \_\_exit\_\_ method and release external resources and that is the reason we do not need finally clause in our code.

Note: There are times when you might need to work with different external resources, say you are reading data from one file and writing it on another.

In that case, we can open another external resource on the same line along *with* statement.

try:

    with open("app.py") as file, open("another.txt") as target:

        print("file opened")

**Raising Exceptions**:

So far we have learned, how to handle exceptions. But we can also raise or throw exceptions in our own code.

def calculate\_xfactor(age):

    return 10 / age

Now age cannot be zero or less, so we can have an if statement here and use ***raise*** keyword to raise and exception like this.

def calculate\_xfactor(age):

    if age <= 0:

        raise ValueError("Age cannot be zero or less")

    return 10 / age

We also need to use try – except block to handle this raised exception,

def calculate\_xfactor(age):

    if age <= 0:

        raise ValueError("Age cannot be zero or less")

    return 10 / age

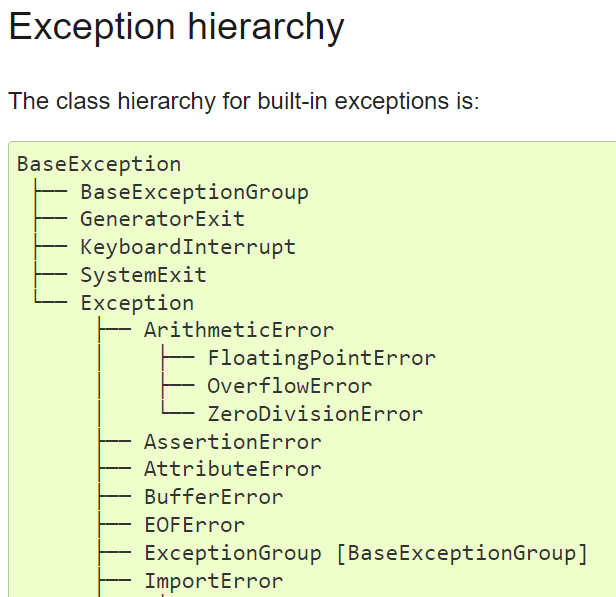
try:

    calculate\_xfactor(0)

except ValueError as error:

    print(error) // Age cannot be zero or less

Note: If you search for Python3 built in exceptions.



We see a hierarchy like a parent – child kind of relationship. They are sufficient for most cases, but you can also define your own exceptions.

**Cost of raising Exceptions**:

When writing your code, prefer not to raise your own exceptions, because it comes with a cost.

To get the time it takes to execute our python code, we can use timeit module.

Then encapsulate your entire code into a single variable like above.

Finally pass *code1* as the first argument of *timeit* and 10000 as a second keyword argument *number*(*number of times we want to execute our code*).

from timeit import timeit

code1 = """

def calculate\_xfactor(age):

    if age <= 0:

        raise ValueError("Age cannot be zero or less")

    return 10 / age

try:

    calculate\_xfactor(0)

except ValueError as error:

    print(error)

"""

print("code1=", timeit(code1, number=10000)) // code1= 1.449593700002879

In second code we will return None object instead of raising exception.

code2 = """

def calculate\_xfactor(age):

    if age <= 0:

        return None

    return 10 / age

try:

    calculate\_xfactor(0)

except ValueError as error:

    print(error)

"""

print("code2=", timeit(code2, number=10000)) // code2= 0.000824799994006753

**Think twice before raising an exception, try to handle situation with a simple if statement**.