# Catching Exception

# Catching Exceptions Using try and except Statement

Error types:

- **Syntax errors**: Errors where the code is not valid Python (generally easy to fix)

- **Runtime errors**: Errors that occur during the execution of the program. Here a syntactically valid code fails to execute, perhaps due to invalid user input (sometimes easy to fix)

- **Semantic errors**: Errors in logic: code executes without a problem, but the result is not what you expect (often very difficult to track-down and fix)

# Syntax Errors

- Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python. For example,

```
1 while True
2    print("Hello World)
```

```
  File "<ipython-input-11-eea1218b3382>", line 1
    while True
               ^
SyntaxError: invalid syntax
```

In the output, the offending line is repeated and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected ①. The error is caused by a missing colon (':'). File name and line number are also printed so you know where to look in case the input came from a Python program file.

# Semantics Error

- Also known as logical error. the program output is not as expected
- **Example**: Consider the following program that print all numbers from 1 to 9 while it is expected to print even numbers from 1 to 9

```
1 for i in range(10):
2     print(i, end=', ')
```

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

# Runtime Errors

For example, if you try to reference an undefined variable:

```
print(Q)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
<ipython-input-3-e796bdcf24ff> in <module>()
----> 1 print(Q)

NameError: name 'Q' is not defined
```

Or you might be trying to compute a mathematically ill-defined result:

```
2 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call
<ipython-input-5-ae0c5d243292> in <module>()
----> 1 2 / 0

ZeroDivisionError: division by zero
```

Or if you try an operation that's not defined:

```
1 + 'abc'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-4-aab9e8ede4f7> in <module>()
----> 1 1 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Or maybe you're trying to access a sequence element that doesn't exist:

```
L = [1, 2, 3]
L[1000]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
<ipython-input-6-06b6eb1b8957> in <module>()
      1 L = [1, 2, 3]
----> 2 L[1000]

IndexError: list index out of range
```

Note that in each case, Python is kind enough to not simply indicate that an error happened, but to spit out a meaningful exception that includes information about what exactly went wrong, along with the exact line of code where the error happened.

# Catching Exceptions: try and except

- The main tool Python gives you for handling runtime exceptions is the try...except clause. Its basic structure is this:

```
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an error")
```

```
this gets executed first
```

- Note that the second block here did not get executed: this is because the first block did not return an error. Let's put a problematic statement in the try block and see what happens:

```
try:
    print("let's try something:")
    x = 1 / 0 # ZeroDivisionError
except:
    print("something bad happened!")
```

```
let's try something:
something bad happened!
```

- Here we see that when the error was raised in the **try** statement (in this case, a **ZeroDivisionError**), the error was caught, and the **except** statement was executed.

# Solve: Zero Division Error Exception

- One way this is often used is to check user input within a function or another piece of code. For example, we might wish to have a function that catches zero-division and returns some other value, perhaps a suitably large number like $10^{100}$:

```
In [9]: def safe_divide(a, b):
            try:
                return a / b
            except:
                return 1E100

In [10]: safe_divide(1, 2)
Out[10]: 0.5

In [11]: safe_divide(2, 0)
Out[11]: 1e+100
```

There is a subtle problem with this code, though: what happens when another type of exception comes up? For example, this is probably not what we intended:

```
safe_divide (1, '2')
```

```
1e+100
```

Dividing an integer and a string raises a **TypeError**, which our code caught and assumed was a **ZeroDivisionError**! For this reason, it's nearly always a better idea to catch exceptions explicitly.

# Catch exceptions explicitly

In [13]:
```
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return 1E100
```

In [14]:
```
safe_divide(1, 0)
```

Out[14]:
```
1e+100
```

In [15]:
```
safe_divide(1, '2')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
<ipython-input-15-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')

<ipython-input-13-10b5f0163af8> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

We're now catching zero-division errors only, and letting all other errors pass through un-modified.

In [26]:

```
1  a=int(input("enter a number: "))
2  b=int(input("enter a number: "))
3  try:
4          print( a / b)
5  except ZeroDivisionError:
6          print( 1E100)
7  except TypeError:
8          print( 'cannot divide int by str')
```

```
enter a number: 3
enter a number: 0
1e+100
```

# Raising Exceptions: raise

- We've seen how valuable it is to have informative exceptions when using parts of the Python language.

- It's equally valuable to make use of informative exceptions within the code you write, so that users of your code (foremost yourself!) can figure out what caused their errors.

- The way you raise your own exceptions is with the raise statement. For example:

```
In [16]: raise RuntimeError("my error message")
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call
<ipython-input-16-c6a4c1ed2f34> in <module>()
----> 1 raise RuntimeError("my error message")

RuntimeError: my error message
```

# Example: Apply raise exception on factorial

```
1  def factorial(N):
2      fact=1
3      for i in range(2, N+1):
4          fact*=i
5      print(fact)
```

```
1  factorial(5)
```

```
120
```

```
1  factorial(-3)
```

```
1
```

- One potential problem here is that the input value could be negative.
- This will not currently cause any error in our function, but we might want to let the user know that a negative **N** is not supported.
- Errors stemming from invalid parameter values, by convention, lead to a **ValueError** being raised

# Example: Apply raise exception on factorial .....

```
In [49]:
def factorial(N):
    if N<0: raise ValueError("Cannot calculate the factorial for a negative number.")
    fact=1
    for i in range(2, N+1):
        fact*=i
    print(fact)
```

```
In [50]:
factorial(5)
```

```
120
```

```
In [42]:
factorial(-3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-42-b34e27388b56> in <module>
----> 1 factorial(-3)

<ipython-input-40-8476e7411778> in factorial(N)
      1 def factorial(N):
----> 2     if N<0: raise ValueError("Cannot calculate the factorial for a negative numbe
r.")
      3     fact=1
      4     for i in range(2, N+1):
      5         fact*=i

ValueError: Cannot calculate the factorial for a negative number.
```

Now the user knows exactly why the input is invalid,
and could even use a **try...except** block to handle it!

# Example: Apply raise exception on factorial .....

Handling the factorial exception

In [52]:
```
try:
    factorial(5)
except ValueError:
    print("Cannot find the factorial for negative values")
```

```
120
```

In [53]:
```
try:
    factorial(-3)
except ValueError:
    print("Cannot find the factorial for negative values")
```

```
Cannot find the factorial for negative values
```

# Accessing the error message

- Sometimes in a **try...except** statement, you would like to be able to **work with** the **error message itself**.
- This can be done with the **as** keyword:

```
1 try:
2     factorial(-3)
3 except ValueError as x:
4     print(x)
```

```
Cannot calculate the factorial for a negative number.
```

```
try:
    x = 1 / 0
except ZeroDivisionError as err:
    print("Error class is:  ", type(err))
    print("Error message is:", err)
```

```
Error class is:   <class 'ZeroDivisionError'>
Error message is: division by zero
```

# Defining custom exceptions

- In addition to built-in exceptions, it is possible to **define custom exceptions** through **class inheritance**. For instance, if you want a special kind of ValueError, you can do this:

```
1 class MySpecialError(ValueError):
2     pass
```

```
do something
do something else
```

- **Raising** the custom Exception

```
1 raise MySpecialError
```

```
---------------------------------------------------------------------------
MySpecialError                            Traceback (most recent call last)
<ipython-input-61-473ba90a0cab> in <module>
----> 1 raise MySpecialError

MySpecialError:
```

- **Handling** the custom Exception using **try...except** block

```
1 try:
2     print("do something")
3     raise MySpecialError("[informative error message here]")
4 except MySpecialError:
5     print("do something else")
```

```
do something
do something else
```

```
1 try:
2     print("do something")
3     raise MySpecialError("[informative error message here]")
4 except :
5     print("do something else")
```

```
do something
do something else
```

# *try...except...else..finally*

- In addition to try and except, you can use the else and finally keywords to further tune your code's handling of exceptions. The basic structure is this:

In [25]:
```
try:
    print("try something here")
except:
    print("this happens only if it fails")
else:
    print("this happens only if it succeeds")
finally:
    print("this happens no matter what")
```

```
try something here
this happens only if it succeeds
this happens no matter what
```

In [63]:
```
1 try:
2     print("try something here")
3     raise ValueError
4 except:
5     print("this happens only if it fails")
6 else:
7     print("this happens only if it succeeds")
8 finally:
9     print("this happens no matter what")
```

```
try something here
this happens only if it fails
this happens no matter what
```

***else* block:** when present, must follow all except blocks.

- It is useful for code that must be executed if the try block does not raise an exception.

***finally* block:** which is intended to define clean-up actions that must be executed under all circumstances.

- A finally block is always executed before leaving the try statement, whether an exception has occurred or not.
- When an exception has occurred in the try block and has not been handled by an except block, it is re-raised after the finally block has been executed.
- The finally clause is also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

# Example1

- Program to Check for ValueError Exception

```
while True:
    try:
        number = int(input("Please enter a number: "))
        print(f"The number you have entered is {number}")
        break
    except ValueError:
        print("Oops! That was no valid number. Try again…")
```

```
Please enter a number: w
Oops! That was no valid number. Try again…
Please enter a number: 3
The number you have entered is 3
```

# Example 2: Trace the following

```
1  x = int(input("Enter value for x: "))
2  y = int(input("Enter value for y: "))
3  try:
4      result = x / y
5  except ZeroDivisionError:
6      print("Division by zero!")
7  else:
8      print(f"Result is {result}")
9  finally:
10     print("Executing finally clause")
```

```
Enter value for x: 2
Enter value for y: 9
Result is 0.2222222222222222
Executing finally clause
```

---

```
Enter value for x: 2
Enter value for y: 0
Division by zero!
Executing finally clause
```

# Example 3

- Write a Program Which Repeatedly Reads Numbers Until the User Enters 'done'.
- Once 'done' Is Entered, Print Out the Total, Count, and Average of the Numbers.
- If the User Enters Anything Other Than a Number, Detect Their Mistake Using try and except and Print an Error Message and Skip to the Next Number

In [73]:

```python
total = 0
count = 0
while True:
    num = input("Enter a number: ")
    if num == 'done':
        print(f"Sum of all the entered numbers is {total}")
        print(f"Count of total numbers entered {count}")
        print(f"Average is {total / count}")
        break
    else:
        try:
            total += float(num)
        except:
            print("Invalid input")
            continue
        #else:count += 1   #remove the previous and next statement
        count += 1
```

```
Enter a number: 3
Enter a number: e
Invalid input
Enter a number: 4
Enter a number: 6
Enter a number: done
Sum of all the entered numbers is 13.0
Count of total numbers entered 3
Average is 4.333333333333333
```