Error handling

In [1]:
```
print("sai")

add = "sai" + 2

print("sai")
```

```
sai

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-d92249210e44> in <module>
      1 print("sai")
      2
----> 3 add = "sai" + 2
      4
      5 print("sai")

TypeError: can only concatenate str (not "int") to str
```

In [2]:
```
div = 67/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-e94ef31e0b2a> in <module>
----> 1 div = 67/0

ZeroDivisionError: division by zero
```

Exceptions handling

In [15]:
```
print("sai")

try:
    add = 3 + "sai"
    print(add)
except Exception as e:
    print("There was an error !!- ", e)

print("sai")
```

```
sai
There was an error !!-  unsupported operand type(s) for +: 'int' and 'str'
sai
```

In [12]:
```
# WAP for writing a file

file = open("fantasticFive.txt","r")

try:
    file.write("HI Guys")
except Exception as e:
    print("There was an error in the file handling ")
    print("Error was - ", e)
finally:
    file.close()

print("Hey guys, rate the session as of right now")
```

```
There was an error in the file handling
Error was -  not writable
Hey guys, rate the session as of right now
```

Unit Testing !!!

- Testing is a part of software development, in which your each simple module is tested.
- To make sure it is enacting the same way for it was designed
- and also to make sure that our code is industry standard

PyLint -

for what reason Pylint is been used, ?

Pylint is a Python static code analysis tool which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions.

In [17]:
```
!pip install pylint
```

```
Requirement already satisfied: pylint in /opt/anaconda3/lib/python3.7/site-packa
ges (2.4.2)
Requirement already satisfied: isort<5,>=4.2.5 in /opt/anaconda3/lib/python3.7/s
ite-packages (from pylint) (4.3.21)
Requirement already satisfied: astroid<2.4,>=2.3.0 in /opt/anaconda3/lib/python
3.7/site-packages (from pylint) (2.3.1)
Requirement already satisfied: mccabe<0.7,>=0.6 in /opt/anaconda3/lib/python3.7/
site-packages (from pylint) (0.6.1)
Requirement already satisfied: lazy-object-proxy==1.4.* in /opt/anaconda3/lib/py
thon3.7/site-packages (from astroid<2.4,>=2.3.0->pylint) (1.4.2)
Requirement already satisfied: wrapt==1.11.* in /opt/anaconda3/lib/python3.7/sit
e-packages (from astroid<2.4,>=2.3.0->pylint) (1.11.2)
Requirement already satisfied: typed-ast<1.5,>=1.4.0; implementation_name == "cp
ython" and python_version < "3.8" in /opt/anaconda3/lib/python3.7/site-packages
(from astroid<2.4,>=2.3.0->pylint) (1.4.1)
Requirement already satisfied: six==1.12 in /opt/anaconda3/lib/python3.7/site-pa
ckages (from astroid<2.4,>=2.3.0->pylint) (1.12.0)
WARNING: You are using pip version 20.0.2; however, version 20.1.1 is available.
You should consider upgrading via the '/opt/anaconda3/bin/python -m pip install
--upgrade pip' command.
```

In [19]:
```
# We will use Magic Function --
```

In [36]:
```
%%writefile treebox.py
'''
A Simple Python program for our testing purpose
'''

A = 1
B = 2

def add(num1, num2):
    '''
    this is a Addition function, which prints the addition of two numbers
    '''
    addition = num1 + num2
    print(addition)

add(A, B)
```

```
Overwriting treebox.py
```

In [35]:
```
! Pylint treebox.py
```

```
-------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 8.33/10, +1.67)
```

In [37]:
```
! Pylint myCode.py
```

```
************* Module myCode
myCode.py:1:0: C0103: Module name "myCode" doesn't conform to snake_case naming
style (invalid-name)

-----------------------------------
Your code has been rated at 8.33/10
```

Unittesting

What is unit testing ?

- unit are the smallest possible block of our program,
- like functions and other stuff,
- we want to test them that they are performing the same manner what we want them to do
- Then we use unit testing --

- AGENDA - to Learn How UNIT TESTIN Works !!!

- Step 1 - Create a new Python File - AreaOfCircle
- Step 2 - Create a UNIT-TESTING file which can test your AreaOfCircle
- Step 3 -

In [ ]:
```

```

In [44]:
```
%%writefile area_of_circle.py
'''
this is a small python file which is used for area of circle
'''
def areaCircle(radius):
    '''
    this is a function for area of circle
    '''
    return 3.14*radius * radius
```

```
Overwriting area_of_circle.py
```

In [59]:
```
%%writefile circle_area_testing.py

import unittest
import area_of_circle

class CircleTesting(unittest.TestCase):
    def testRadius(self):  # TestCase 1
        temp_radius = 4
        result = area_of_circle.areaCircle(temp_radius)
        #Important line
        self.assertAlmostEqual(result,50.24)
        # THe above line checks your result from function is equal to
        # the manully given result,
        # if yes then it says test passed
        # if no then it says test failed

    def testRadiusTwo(self): # TestCase 2
        temp_radius = 101
        result = area_of_circle.areaCircle(temp_radius)
        self.assertAlmostEqual(result,32031.14)

if __name__ == "__main__":
    unittest.main()
```

```
Overwriting circle_area_testing.py
```

In [60]:
```
! python circle_area_testing.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

In [56]:
```
import area_of_circle

area_of_circle.areaCircle(4)
```

Out[56]:
```
50.24
```

In [ ]: