Exception Handling and Assertions: Basics

INDRAPRASTHA INSTITUTE *of* INFORMATION TECHNOLOGY
DELHI

Error and Exceptions

- **Compile Time Error:** A program may have syntax errors - i.e. the statements violate the syntax of Python
  - Syntax errors are detected during compiling
  - Program is not executed
  - These are commonly called *errors*
- **Runtime Errors are Exceptions:** Even if syntax is correct, the program may have errors
  - The program may produce a wrong output/result (no error detected by python - user determines output as wrong)
  - Error may be encountered during execution and execution cannot continue - these runtime errors are *exceptions*
  - Python runtime will give a label to this runtime error - which specifies the type of error it encountered
- Generally both are called Errors

Runtime Errors

- If program passes syntax checks, it can be executed
- During execution an error occurs that disallows further execution - interpreter stops execution
- An error message displayed - this is a runtime error or exception
- Eg: num is a string, so at num%2 runtime raises TypeError

```
num = input()
if num%2==0:
    print("Even")
else:
    print("Odd")
```

```
5
Traceback (most recent call last):
  File "<string>", line 2, in <module>
TypeError: not all arguments converted
during string formatting
```

Runtime Errors examples..

```
def divide(x, y):
    return x/y
def demo():
    divide(2, 0)
def test():
    demo()
test()
```

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
File "<string>", line 6, in test
  File "<string>", line 4, in demo
  File "<string>", line 2, in divide
ZeroDivisionError: division by zero
```

```
def getter(x, i):
    return x[i]
```

```
print(getter("demo",3)) # No error!
```

Output:

```
o
```

```
print(getter("demo",4)) # Runtime error!
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
File "<string>", line 2, in getter
IndexError: string index out of range
```

Built-in Exceptions (runtime Errors)

- Many built-in exceptions - most of them are named as error (they are errors which occur during runtime)
- Each exception also specifies the condition when it is raised. Some of these are groups of exceptions. Eg.
  - **IndexError:** when index is out of bounds
  - **KeyError:** When key not found in dictionary
  - **NameError:** When a local var is used but no value assigned
  - **ZeroDivisionError:** Divide by 0 encountered
  - **ValueError:** Operation receives a value that is not appropriate
  - **IOError:** Wrong file name or path while reading
  - **RuntimeError:** A general error when none fits the case
- Exceptions can be "caught" in a program, and some action taken in program.
- Users can also define exceptions and raise them

Quiz – Single Correct

Suppose the directory in which the program is, does not have a file called "hello". Which exception is raised on executing:

f = open("hello")

A. ValueError
B. NameError
C. FileNotFoundError
D. KeyError

Quiz – Single Correct

Suppose the directory in which the program is does not have a file called "hello". Which exception is raised on executing:

```
f = open("hello")
```

A. ValueError
B. NameError
C. FileNotFoundError
D. KeyError

Handling Exceptions

- In some block of code, if an exception occurs, python provides the ability to "catch" it and execute some code
- With this, instead of the program stopping and printing the exception, it will execute the "handler" code provided
- As you may want to take different actions for different parts of the code, the "exception handler" is attached to a block of code
- This is provided by the ***try*** statement, which has ***except*** clause to specify the exceptions to be caught and code for handling them
- Sometimes called the try-except block; in other languages try-catch block is used for exception handling
- Use-case example: Server programs need to remain active even after internal errors!

Execution of try–except block

- First block of code of try (i.e. between try and except) is executed
- If no exception occurs, the except part is skipped
- If an exception occurs in any statement in the try block, execution of rest of the try block is skipped
- If there is a except statement for the raised error, then the code block of except is executed
- Execution continues after the try-except statement
- If no except block provided for the error, attempt is made in the enclosing try-block, if there is one
- If not, this is unhandled exception, and python does what it would if there was no try-except statement

Example

```
def getter(x, i):
    return x[i]
```

```
print(getter("demo",3)) # No error!
```

Output:

```
o
```

```
print(getter("demo",4)) # Runtime error!
```

Output:

```
Traceback (most recent call last):
  File "<string>", line 4, in <module>
File "<string>", line 2, in getter
IndexError: string index out of range
```

```
def getter(x, i):
    return x[i]
try:
    s, i = "demo", 3
    s = "demo"
    print(getter(s,i))
    i = 4
    print(getter(s,i))
except IndexError:
    if i>=len(s):
        print("Index more than len")
    elif i<0:
        print("Index less than 0")
print("Continuing ...")
```

Output:

```
o
Index more than len
Continuing ...
```

Example

If you take input and convert it to int, you get a ValueError if input is not int. Normally, python will print an error message and stop execution, e.g.

```
# give 3.5 as input
x = int(input())


Output:
Traceback (most recent call last):
    File  "<pyshell#35>",  line  1,  in
<module>
    x = int(input())
ValueError:   invalid   literal   for
int()...
```

# Program to catch ValueError and ask the user to try again.

```
while True:
  try:
     x = int(input("Input:"))
     break
  except ValueError:
     print("Incorrect input - Try again")
print("Input is: ", x)

Output:
Input:3.5
Incorrect input - Try again
Input:3
Input           is:                 3
```

Examples

```
# Example of try except
arr = [1, 2, 3, 4, 5]

try:
    index = int(input())
    print(arr[index])
except IndexError:
    print("Index out of range")
```

```
Input
3
Output
4
```

```
Input
30
Output
Index out of range
```

General except clause

- Except clause does not need to specify exception names
- In this case, it will be executed for all exceptions

```
try:
        res = 5/0
except:
        print("Exception ...")
```

- When multiple except statements, this has to be the last
    - Normally, this is used as the default handler to handle unhandled errors
- Not a good practice to use this for a "general handler", which is not feasible mostly
- Use it only after having handlers for most common exceptions

Quiz (Multi-option correct)

Which of the following represents the correct usage of try-except block?

A)

```
try:
        file=open("name.py")
except:
        print("File not found")
```

B)

```
try:
        file=open("name.py")
except:
        print("File not found")
except:
        print("Unknown error occured")
```

C)

```
try:
        file=open("name.py")
try:
        file=open("file.py")
except:
        print("File not found")
```

D)

```
try:
        file=open("name.py")
print(file)
except:
   print("file not found")
```

Quiz (Solutions)

Which of the following represents the correct usage of try-except block?

A)

```
try:
        file=open("name.py")
except:
        print("File not found")
```

B)

```
try:
        file=open("name.py")
except:
        print("File not found")
except:
        print("Unknown error occured")
```

C)

```
try:
        file=open("name.py")
try:
        file=open("file.py")
except:
        print("File not found")
```

D)

```
try:
        file=open("name.py")
print(file)
except:
   print("file not found")
```

Options A and B are correct

Exercise

- Often we write programs that ask user to give a file name
- If any typo - the program crashes - IOError (or FileNotFoundError)
- Write code to read a file, and prompt user again if a typo, rather than crashing (just one more try), and print contents of file

```
fname = input("File Name: ")

…


print(f.read())
```

Examples...

# Get file name, open it, if error prompt user to give a diff name

```
gotFile = False
while (gotFile==False):
    try:
        fname = input("Give File Name: ")
        f = open(fname)
        gotFile=True
    except IOError:
        print(f'could not open {fname}; Try again')
```

# Read file as integers to sum - if data not int, just discard it

```
tot, num, errnum = 0, 0, 0
for line in f:
    line = line.split()
    for elt in line:
        try:
            tot += int(elt)
            num += 1
        except ValueError:  # the elt is not an integer
            errnum += 1

print(f'Item: {num}; Total: {tot}; Error numbers: {errnum}')
f.close()
```

Summary – Exception Handling

- Main error types: Syntax(compile), logic (runtime - user has to determine), runtime error
- Runtime errors - exceptions are raised by python; default we get a traceback error message - tells where the error is, its type, …
- We can catch raised exceptions in our program, and do something to avoid the program from "crashing"
    - Important for "always-running" software
- Done by try-except block - if any stmt in try block generates a exception, transfer goes to handler for that exception for this block
- There is more - you can have multiple handlers, default handlers, define your own exceptions, etc…

Assert statement

- Assert statements are usually used for debugging and testing code during software development
- They state a condition (‘assert’) which should hold at some point in the program
- These are boolean expressions that evaluate if the conditions return True or False
- Your thought process should be:

  “I need to ensure this condition should remain true. Otherwise, I need to throw an Error”
- Usage:

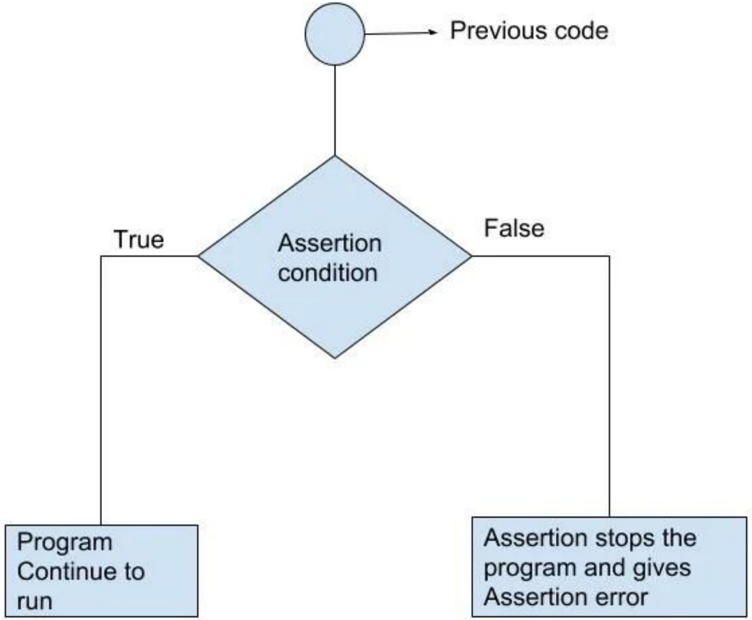
```
assert <condition>
```

(or)

```
assert <condition>,<error_message>
```

Assert statement

- When the assert condition arrives,
- If the assert is True,
    - The program continues running…
- If the assert is False,
    - The program halts, and throws AssertionError



Source: Programiz

Uses / purpose of assert

- Assert statements are widely used in testing and debugging
- We can use it for sanity checks i.e. for testing if a particular assumption is True while writing code
- If it becomes False, then there is a bug in the code.
  - After testing, we can remove assert statements - there is also a compile time option to ignore the assert statements
- This makes our code more robust, reliable, and less prone to errors and bugs
- Also used for writing test cases for code

- **Warning:** Assert is **NOT** an error handling tool - so you don’t need to provide handlers for it

  (Use try-except statement for that)

Examples of Assert

```
def division(a, b):
    assert b != 0, “Cannot put b as 0 since we can’t divide by 0”
    return a / b

x, y = input().split()
print(division(int(x), int(y)))
```

```
Input
10 2
Output
5.0
```

```
Input
4 0
Output
Traceback ...
AssertionError: Cannot put b as 0 since we can’t divide by 0
```

Examples of Debugging with Assert

```
def rectangle_area(l, b):
    assert l > 0, "Length should be positive"
    assert b > 0, "Breadth should be positive"
    return l*b

x, y = input().split()
print(rectangle_area(int(x), int(y)))
```

```
Input
10 2
Output
20.0
```

```
Input
-1 5
Output
Traceback ...
AssertionError: Length should
be positive
```

```
Input
4 -6
Output
Traceback ...
AssertionError: Breadth should be
positive
```

Quiz – Single correct

Which of the options is the output of the code given on the right?

A. **`Assertion Error`**

B. **`Zero Division`**

C. **`Assertion Error`**

   `Zero Division Error`

D. **`Here!`**

```
def test(val):
    try:
        assert val != 0
        print(10 / val)
        print("Here!")

    except AssertionError:
        print("Assertion Error")

    except ZeroDivisionError:
        print("Zero Division Error")

test(0)
```

Quiz – Single correct(Solution)

Which of the options is the output of the code given on the right?

A. **Assertion Error**

B. **Zero Division**

C. **Assertion Error**
   Zero Division Error

D. **Here!**

```
def test(val):
    try:
        assert val != 0
        print(10 / val)
        print("Here!")

    except AssertionError:
        print("Assertion Error")

    except ZeroDivisionError:
        print("Zero Division Error")

test(0)
```

Purpose of Asserts

- It provides user defined mechanism for identifying errors during execution - assert conditions are based on the code design
- They help designers in designing - you have to think carefully what conditions must hold at some point
  - Example - in while loop for computing something you can assert something about the computation - some value is increasing/decreasing
- Used in testing - test cases are written in a program - each test case gives inputs and checks for correct output through assert stmt
- Generally in production code, assert statements are disabled (to avoid the overheads of these checks)
  - This makes it more useful than checking these conditions in if stmt
- I.e. assert are not for error handling during runtime of a production system, use try-except block for this
- Many practices propagate the use of asserts during programming

Automated Testing using Assert

- A function/program to compute something
- Write testcases as functions - they set the test data, call the function, check the value using assert
- A testcase script to run these testcase functions - if any of them fails, user gets a notification
- Can be used for automated test scripts - this is what unittest, pytest use in some form

Examples

Lets see some examples

Summary

- Asserts are used to in-line test programs
- Often use during testing and program development
- You can check for value (==), for membership (using in, not in), type of objet (using isinstance(), type()), comparison (relational ops),
- But can cause overhead in the final software - there are ways to give directive that asserts are not executed (so no overhead in execution)
- Used extensively in unitesting frameworks like unittest, pyunit which allow programmers to write testing scripts