Error and Exception Handling

Debugging

Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

- Syntax errors
- Runtime errors
- Semantic errors

Now we are considering the same example what we have discussed in last class:

```
# Example:  create a static method using staticmethod ()
class Maths:
    def addnumbers(x,y):
        return x+y
 # create addnumbers static method
Maths.adnumbers = staticmethod(Maths.addnumbers)
print(" The sum is:", Maths.addnumbers(5,10))
```

```
 The sum is: 15
```

Before going to discuss different types of Errors in Python, go through with the below statements and if you miss to maintain the below statements during your program execution you may end up with the below errors

Slide 1: Syntax Errors

Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.

2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.

3. Check that indentation is consistent. You may indent with either spaces or tabs but its best not to mix them. Each level should be nested the same amount.

4. Make sure that any strings in the code have matching quotation marks.

5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

6. An unclosed bracket – (, {, or [– makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

7. Check for the classic = instead of == inside a conditional.

Some of the different types of Errors we may come across:

1. **Assertion Error:** Raised when the assert statement fails.
2. **AttributeError:** Raised on the attribute assignment or reference fails.
3. **EOFError:** Raised when the input () function hits the end-of-file condition.
4. **FloatingPointError:** Raised when a floating point operation fails.
5. **GeneratorExit:** Raised when a generator's close () method is called.
6. **ImportError:** Raised when the imported module is not found.
7. **IndexError:** Raised when the index of a sequence is out of range.
8. **KeyError:** Raised when a key is not found in a dictionary.
9. **KeyboardInterrupt:** Raised when the user hits the interrupt key (Ctrl+c or delete).
10. **MemoryError:** Raised when an operation runs out of memory.
11. **NameError:** Raised when a variable is not found in the local or global scope.
12. **NotImplementedError:** Raised by abstract methods.
13. **OSError:** Raised when a system operation causes a system-related error.
14. **OverflowError:** Raised when the result of an arithmetic operation is too large to be represented.
15. **ReferenceError:** Raised when a weak reference proxy is used to access a garbage collected referent.
16. **RuntimeError:** Raised when an error does not fall under any other category.
17. **StopIteration :** Raised by the next () function to indicate that there is no further item to be returned by the iterator.
18. **SyntaxError:** Raised by the parser when a syntax error is encountered.
19. **IndentationError:** Raised when there is an incorrect indentation.
20. **TabError:** Raised when the indentation consists of inconsistent tabs and spaces.
21. **SystemError:** Raised when the interpreter detects internal error.
22. **SystemExit:** Raised by the sys.exit () function.

23. **TypeError:** Raised when a function or operation is applied to an object of an incorrect type.
24. **UnboundLocalError** : Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
25. **UnicodeError:** Raised when a Unicode-related encoding or decoding error occurs.
26. **UnicodeEncodeError:** Raised when a Unicode-related error occurs during encoding.
27. **UnicodeDecodeError:** Raised when a Unicode-related error occurs during decoding.
28. **UnicodeTranslateError:** Raised when a Unicode-related error occurs during translation.
29. **ValueError:** Raised when a function gets an argument of correct type but improper value.
30. **ZeroDivisionError:** Raised when the second operand of a division or module operation is zero.

Syntax errors: are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program.

```
# Example:  create a static method using staticmethod ()
class Maths:
    def addnumbers(x,y)
        return x+y
 # create addnumbers static method
Maths.adnumbers = staticmethod(Maths.addnumbers)
print(" The sum is:", Maths.addnumbers(5,10))
```

```
  File "<ipython-input-10-00db335b6ee5>", line 3
    def addnumbers(x,y)
                      ^
SyntaxError: invalid syntax
```

Example: Omitting the colon at the end of a def statement yields the somewhat redundant message **SyntaxError: invalid syntax.**

NameError: Raised when a variable is not found in the local or global scope.

```
# Example:  create a static method using staticmethod ()
class Maths:
    def addnumbers(x,y):
        return xy
 # create addnumbers static method
Maths.adnumbers = staticmethod(Maths.addnumbers)
print(" The sum is:", Maths.addnumbers(5,10))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-20-3786a0327a8e> in <module>
      5  # create addnumbers static method
      6 Maths.adnumbers = staticmethod(Maths.addnumbers)
----> 7 print(" The sum is:", Maths.addnumbers(5,10))

<ipython-input-20-3786a0327a8e> in addnumbers(x, y)
      2 class Maths:
      3     def addnumbers(x,y):
----> 4         return xy
      5  # create addnumbers static method
      6 Maths.adnumbers = staticmethod(Maths.addnumbers)

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

In the above program the return statement must be **"return x+y"**, but + operator is missing hence we got the error: **NameError:** name '**xy**' is not defined

IndentationError: Raised when there is an incorrect indentation.

Example:

```
class Maths:
    def addnumbers(x,y):
        return x+y
```

After modification in the code:

```
class Maths:
    def addnumbers(x,y):
    return x+y
```

```
  File "<ipython-input-29-9952aaaf47ae>", line 3
    return x+y
           ^
IndentationError: expected an indented block
```

Assignment: Apart from the above example given try yourself with different types of errors so that you will become perfect in handling errors in python.

Python Exception:

- If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.
- The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked *that*, and so on.

Example: Before going to learn about python Exception let us understand what is traceback?

- Python prints a **traceback** when an exception is raised in your code.
- The traceback output can be a bit overwhelming if you're seeing it for the first time or you don't know what it's telling you.
- But the Python traceback has a wealth of information that can help you diagnose and fix the reason for the exception being raised in your code.
- Understanding what information a Python traceback provides is vital to becoming a better Python programmer.

What Is a Python Traceback?

- A traceback is a report containing the function calls made in your code at a specific point. Tracebacks are known by many names, including **stack trace, stack traceback, backtrace**, and maybe others. **In Python, the term used is traceback.**
- When your program results in an exception, Python will print the current traceback to help you know what went wrong. Below is an example to illustrate this situation:

```
# example.py
def greet(someone):
    print('Hello, ' + someon)

greet('Chad')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-36-29a798210489> in <module>
      3     print('Hello, ' + someon)
      4
----> 5 greet('Chad')

<ipython-input-36-29a798210489> in greet(someone)
      1 # example.py
      2 def greet(someone):
----> 3     print('Hello, ' + someon)
      4
      5 greet('Chad')

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

```
""" in the above example, greet() gets called with the parameter someone. However,
in greet(), that variable name is not used. Instead, it has been misspelled as someon
in the print() call."""
```

For more details on Python Traceback: find the below web link:

https://realpython.com/python-traceback/#python-traceback-overview

Why use Exceptions?

Exceptions are convenient in many ways for handling errors and special conditions in a program. When you think that you have a code which can produce an error then you can use exception handling.

The problem without handling exceptions:

As we have already discussed, the exception is an abnormal condition that halts the execution of the program from above examples.

Suppose we have two variables a and b, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a ZeroDivision exception. Let's see the following example.

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)

#other code:
print("Hi I am other part of the program")
```

```
Enter a:1
Enter b:0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-1-a104c1c193bd> in <module>
      1 a = int(input("Enter a:"))
      2 b = int(input("Enter b:"))
----> 3 c = a/b
      4 print("a/b = %d" %c)
      5

ZeroDivisionError: division by zero
```

Now in order to handle the problem of Exceptions in our programs we can make use of below statements to clear the exceptions and allow the program to execute with proper message for different errors in the python script.

- The **"Try"** block lets you test a block of code for errors.
- The **"Except"** block lets you handle the error.
- The **"Finally"** block lets you execute code, regardless of the result of the try- and except blocks.

Below examples will give the idea of how the "Try", "Except/Exception" and "Finally" works with their feature

Exception Handling

When an error occurs or exception, Python will normally stop and generate an error message. These exceptions can be handled using the try statement:

Example

The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

```
An exception occurred
```

Explanation:

- Since the try block raises an error, except block will be executed.
- Without the try block, the program will crash and raise an error
- If you want to print 'x' some where it has to be declared with some value but we have not done that and directly running our code hence we end up with the Exception message.

```
# Next Example
print(x)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-fc17d851ef81> in <module>
----> 1 print(x)

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

```
"in the above example The " print" statement will raise an error, because x is not defined"
```

```
# Next Example
"""  Many Exceptions
You can define as many exception blocks as you want,
e.g. if you want to execute a special block of code for a special kind of error:
Example: Print one message if the try block raises a NameError and another for other errors:"""
```

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

```
Variable x is not defined
```

```
# Next Example
""" Else: You can use the else keyword to define a block of code to be executed if no errors were raised:
Example: In this example, the try block does not generate any error:"""
```

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

```
Hello
Nothing went wrong
```

```
# Next Example
""" Finally: The finally block, if specified, will be executed regardless if the try block raises an error or not."""
```

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

```
Something went wrong
The 'try except' is finished
```

```
# Next Example: Raise an exception
""" As a Python developer you can choose to throw an exception if a condition occurs.
To throw (or raise) an exception, use the raise keyword. """
```

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-12-2edc57024fbc> in <module>
      2
      3 if x < 0:
----> 4   raise Exception("Sorry, no numbers below zero")

Exception: Sorry, no numbers below zero
```

```
""" From the above example The raise keyword is used to raise an exception.
You can define what kind of error to raise, and the text to print to the user in the below example."""
# Example :Raise a TypeError if x is not an integer in the below code:
```

```
x = "hello"

if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-13-bc91768a6271> in <module>
      2
      3 if not type(x) is int:
----> 4   raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed
```

True or False

1. Syntax errors arises due to poor understanding of the language: **True**
2. Logic errors can be detected by a python interpreter: **False**
3. Even if a statement is syntactically correct, it may still cause an error when executed: **True**
4. An exception disrupts the normal flow of the program's instructions: **True**
5. Standard exception names are reserved words in python: **False**
6. If an exception occurs, during execution of any statement in the try block then, rest of the statements in the try block are skipped: **True**
7. Exceptions gives you information like what, why, and how something went wrong: **True**
8. Python allows you to have multiple except blocks for a single try block: **True**
9. It is possible to execute more than one except block during the execution of the program: **False**
10. No code should be present between the try and except block: **True**
11. You should make extensive use of except block to catch any type of exception that may occur: **False**
12. Else block must follow all except blocks: **True**
13. Else block has statements to handle an exceptions raised from the try block: **False**
14. AttributeError exception is raised when an identifier is not found in local or global namespace: **False**
15. An exception can be a string, a class, or an object: **True**

MCQ: Multiple Choice Questions

1. Which type of error specifies all those type of errors in which the program executes but gives incorrect results?

- Syntax
- **Ans:** Logic
- Exception
- None of these

2. Which keyword is used to generate an exception?

- Throw
- **Ans:** Raise
- Generate
- Try

3. Which block acts as a wildcard block to handle all exceptions?

- Try
- Catch
- Except exception
- Ans: Except

4. Which block is executed when no exception is raised from the try block?

- Try
- **Ans:** Else
- Except exception
- Except

5. To handle an exception, try block should be immediately followed by which block?

- Finally
- Catch
- Else
- **Ans:** Except

6. Which exception is raised when two or more data types are mixed without force?

- **Ans:** TypeError
- AttributeError
- ValueError
- NameError

7. Which block can never be followed by except block?

- **Ans:** Finally
- Catch
- Else
- Except

8. You cannot have which block with a finally block?

- Try
- Catch
- **Ans:** Else
- Except

9. Which statement raises exception if the expression is false?

- Throw
- Else
- Raise
- **Ans:** Assert

10. '1' = = 1 will result in _____?

- True
- Ans: False
- TypeError
- ValueError

11. Which number is not printed by this below code?

Try:

    Print (10)

    Print (5/0)

    Print (20)

Except ZeroDivisionError:

    Print (30)

Finally:

    Print (40)

- **Ans:** 20
- 40
- 30
- 10

Review Questions

1. Difference between error and exception?
2. What are logic errors, give examples?
3. What happens when an exception is raised in a program?
4. What will happen if an exception occurs but is not handled by the program?
5. How can you handle exceptions in your program?
6. Explain the syntax of try-except block?
7. What happens if an exception occurs which does not match the exception named in except block?
8. How can you handle multiple exceptions in a program?
9. Using except block is not recommended. Justify the statement?
10. When is the else block executed?
11. With the help of an example explain how can you instantiate an exception?
12. Explain any three built-in exceptions with relevant examples?
13. How can you create your own exceptions in python?
14. What will happen if an exception generated in the try block is immediately followed by a finally block?
15. Explain the utility of assert statement?