# **Week 9: Lab pre-read**

## 1. Error handling

#### 1.1 Errors

Errors in a language occur when something unexpected happens causing the code to stop execution. We have already learned to categorise errors into two types - **SyntaxError** and **SemanticError**.

Syntax errors occur when the proper syntax or the grammatical construct of the language is not followed while writing code such as:

if university == 'Plaksha'

print('Punjab')

The above code raises the following error:

File "main.py", line 1

if university == 'Plaksha'

^

SyntaxError: invalid syntax

Here the ‘^’ indicates where the interpreter ran into the error while parsing the statement. The syntax of the if statement requires a colon ‘:’ at the end of the statement which leads to this error. You may notice that even though the variable university has not been defined and assigned a value before running the if statement, the error raised is a **SyntaxError** and not a **NameError** (NameError: name 'university' is not defined) which is an exception, pointing to the hierarchy of error checking in Python. This brings us to the concept of exceptions.

#### 1.2 Exceptions

Exception checks are performed once the code has cleared the syntax checks and point to the logical or semantic flaws in the written code which cause the code to stop executing. In other words, **SematicErrors** in python are caught at runtime and relayed to the user in the form of exceptions. A few examples of the common exceptions are:

* **NameError -** when a variable name (local or global) is not found while parsing.

if university == 'Plaksha':

print('Punjab')

Output:

Traceback (most recent call last):

File "main.py", line 1, in <module>

if university == 'Plaksha':

NameError: name 'university' is not defined

* **IndexError -** when the index for a given sequence is out of range for that sequence.

numbers = [1, 2, 3]

numbers[3]

Output:

Traceback (most recent call last):

File "main.py", line 2, in <module>

numbers[3]

IndexError: list index out of range

* **ZeroDivisionError -** when a division is being performed by 0.

for i in range(0, 10):

print(i\*\*(2/i))

Output:

Traceback (most recent call last):

File "main.py", line 2, in <module>

print(i\*\*(2/i))

ZeroDivisionError: division by zero

* **TypeError -** when an inappropriate data type is being operated on.

for i in range(0, 10):

print(i + str(i))

Output:

Traceback (most recent call last):

File "main.py", line 2, in <module>

print(i + str(i))

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

Other examples of exceptions are **ImportError**, **MemoryError**, **AssertionError**, **OSError**, **ModuleNotFoundError** etc. As you go through the various coding exercises you are encouraged to look out for these errors and understand what causes them.

#### 1.3 Exception handling

Unlike a syntax error, exceptions do not necessarily have to lead to the halting of the program execution. There may be parts of your code that require a user input where the user may enter the incorrect file format, or a division statement that can potentially lead to a zero division error being raised or many more such cases wherein you could expect exceptions being raised while writing the code. It then becomes the programmer’s responsibility to write their code keeping these possibilities in mind to *handle* exceptions.

The following code has a semantic flaw:

monthlySalary = 50000

investedAmount = int(input())

Ratio = monthlySalary/investedAmount

print('The investment ratio for this month is:', Ratio)

print('Next month')

When this code is executed it will run properly in most cases except if the user decides to not invest any money for a month. Hence, the program will fail when *investedAmount* is zero. Python will raise a **ZeroDivisionError** exception, and the print statements will never be executed. But this is a type of exception that should be expected. A better way to write this code with exception handling is:

monthlySalary = 50000

investedAmount = int(input())

try:

Ratio = monthlySalary/investedAmount

print('The investment ratio for this month is:', Ratio)

except ZeroDivisionError:

print('No investments made this month')

print('Next month')

Here, in the **try** block the interpreter executes the statements as long as they are evaluated successfully and then executes the print statement. However, if a **ZeroDivisionError** exception is raised, the interpreter moves to the **except** block that catches that specific exception and executes the statement within. In this case, on encountering a **ZeroDivisionError** the print statement saying ‘No investments made this month’ is executed. Finally the interpreter executes the print statement outside the **try-except** block. In this way the potential exception that could be raised has been handled and will not cause the program to halt execution. You can extend this idea of exception handling to many use cases.

##### 1.3.1 Handling multiple exceptions in the same try-except block

A try-except block can also contain multiple exception clauses depending on the context of the code and the possibilities of encountering multiple exceptions. Consider the following code:

def SomeAlgo(temp):

return (100 / temp) - 32

The code has the possibility of encountering multiple exceptions namely **ZeroDivisionError** (if the value of temp is 0) and **TypeError** (if the data type of temp is anything apart from an int or float). Handling both exceptions can be written in the following way:

def SomeAlgo(temp):

try:

return (100 / temp) - 32

except ZeroDivisionError:

print("Enter a non-zero number and try again")

except TypeError:

print("Enter a number as the input and try again")

Furthermore, multiple exceptions can also be handled in the same except block by adding them in a tuple separated by a comma:

def SomeAlgo(temp):

try:

return (100 / temp) - 32

except (ZeroDivisionError, TypeError):

print("Enter a valid input (non-zero number) and try again")

##### 1.3.2 Updating/changing the message of an exception

Exceptions can also be used to control how a piece of code flows. Taking the same example as before, if we want to add specific conditions for the input, let’s say we want the input to only be an integer and not a float, the code can include a ***raise*** statement to raise an exception with a custom message describing what went wrong if the input is a float.

def SomeAlgo(temp):

try:

if type(temp) == float:

raise ValueError(“Enter an integer and try again”)

else:

return (100 / temp) - 32

except (ZeroDivisionError, TypeError):

print("Enter a valid input (non-zero number) and try again")

##### 1.3.3 Custom/user-defined exceptions

You can also have custom or user-defined exceptions that can be raised when certain conditions in your program are not being met the way you might expect them to for ideal execution. For this, you have to create a new exception class that inherits from the built-in *Exception* class of Python either directly or indirectly.

For example, in the code below, we define three new exception classes ***ValueInvalid, ListTooSmall*** and ***ListTooLarge*** that inherits the built-in *Exception* class and are used in the program that follows:

import numpy as np

class ValueInvalid(Exception):

'''Raised if an element in the list is invalid for the expected task'''

pass

class ListTooSmall(Exception):

'''Raised when the list given is smaller than expected'''

pass

class ListTooLarge(Exception):

'''Raised when the list given is larger than expected'''

pass

def calcAvgMarks(marks):

try:

if len(marks) < 5:

raise ListTooSmall

elif len(marks) > 5:

raise ListTooLarge

elif len([x for x in marks if x < 0]) > 0:

errorVal = [x for x in marks if x < 0]

raise ValueInvalid

else:

print(np.average(marks))

except ListTooSmall:

print('Invalid input, please check and try again, number of elements in list is less than 5:', len(marks))

except ListTooLarge:

print('Invalid input, please check and try again, number of elements in list is more than 5:', len(marks))

except ValueInvalid:

print('Check the input again, there seems to be a negative entry:', errorVal)

print("Done!")

Here we have defined three new exception classes, each of which are checking the validity of the input data and constraining the input list to have exactly a length of 5 and only positive values, ensuring smooth running of the program as intended. In this way the context of the program you want to write can be constrained using custom exceptions.

#### 1.4 Assertions

**Assertions** in python are used for debugging the code to ensure a smooth quality assurance process. Assertions are written from the prior knowledge of a programmer that certain conditions have to be met before allowing the code to execute further.

Assertion conditions evaluate to true or false. If the condition is true, the program continues the execution and moves on to the next line of code. However, if it evaluates to false, the program stops by throwing an error.

Let's define a function named check\_threshold to check and print the value of parameter “b” if the value is less than 200, else it throws an assertion error message stating that “**value is out of range**”.

def check\_threshold(b):

print("checking the condition")

assert b < 200, "**value of b is more than 200**"

print(“The value of b is :”, b)

return None

In the above function assert evaluates the condition b < 200, if true, it continues and prints the value of b. Else, it would stop the program with an error message "**value of b is more than 200**".

**Syntax:** assert **condition**, “error\_message(optional)”

Example:

def division(a, b):

print("The value of a / b is:")

assert b != 0, "Zero Division Error"

print(a / b)

**division(4,0)**

Output:

The value of a / b is :

AssertionError: Traceback (most recent call last)

----> 1 division(4,0)

**1** def division(a,b):

**2** print("The value of a / b is : ")

----> 3 assert b != 0, "Zero Division Error"

**4** print(a / b)

AssertionError: Zero Division Error

## 2. Exercise

This programming exercise builds on top of last week’s programming assignment. The students would begin by exchanging their codes (either on Palindrome detection or on Anagram detection) with their partners from the last lab session. They must begin by updating their local repository to the remote repository of their partner in order to have the final version of their partner’s code. The exercise starts now.

1. Peruse the code and identify any **SyntaxError** that may be present. Correct each such error in a separate commit. Note that the commit command should contain a proper message providing textual details of the syntax error that was corrected.
2. Next, run the code to verify that no **SyntaxError** remains. Repeat Step (a) above until all **SyntaxErrors** are corrected.
3. Next, run the code with different input values to **see if the code works correctly i.e. it returns the right answer for each input value(s)**. If not, identify where the bug (an instruction or set of instructions that is not performing as expected or desired) lies. Once again, make the appropriate changes, and only put related changes in one single commit along with the relevant message.
4. Finally, run the code with arbitrary input values to **get the code to throw an exception**. Record these exceptions, and **incorporate exception handling code** to prevent the code from halting during execution. Commit each exception that you have handled separately and with the appropriate textual message.

Push all your commits to your partner’s remote repository. The last 15-20 mins of the class may be spent in each student returning to their own remote repository and pulling the changes made by their partners. Go through the changes and see how different the code looks from the perspective of a peer.

Grade your partner [here](https://forms.office.com/r/YiAHecqCxy).