UNIT - 5

Errors and Exceptions: Syntax Errors, Exceptions, Handling Exceptions, Raising Exceptions, User-defined Exceptions, Defining Clean-up Actions, Redefined Clean-up Actions.

Graphical User Interfaces: The Behavior of Terminal Based Programs and GUI -Based, Programs, Coding Simple GUI-Based Programs, Other Useful GUI Resources.
Programming**:** Introduction to Programming Concepts with Scratch.

^^

https://realpython.com/python-exceptions/

https://www.brianheinold.net/python/python_book.html#chapter_oop

Syntax Errors:

When you run your Python code, the interpreter will first parse it to convert it into Python byte code, which it will then execute. The interpreter will find any invalid syntax in Python during this first stage of program execution, also known as the **parsing stage.** If the interpreter can't parse your Python code successfully, then this means that you used invalid syntax somewhere in your code. The interpreter will attempt to show you where that error occurred.

When the interpreter encounters invalid syntax in Python code, it will raise a **SyntaxError** exception and provide a traceback with some helpful information to help you debug the error.

Ex:

```
1  #ages.py
2   ages = {
3   'pam': 24,
4   'jim': 24
5   'michael': 43
6    }
7   print(f'Michael is {ages["michael"]} years old.')
```

You can see the invalid syntax in the dictionary literal on line 4. The second entry, 'jim', is missing a comma. If you tried to run this code as-is, then you'd get the following traceback:

```
$ python ages.py
File "ages.py", line 5
   'michael': 43
          ^
SyntaxError: invalid syntax
```

There are a few elements of a SyntaxError traceback that can help you determine where the invalid syntax is in your code:

- **The file name** where the invalid syntax was encountered
- **The line number** and reproduced line of code where the issue was encountered
- **A caret (^)** on the line below the reproduced code, which shows you the point in the code that has a problem
- **The error message** that comes after the exception type SyntaxError, which can provide information to help you determine the problem

There are two other exceptions that you might see Python raise. These are equivalent to SyntaxError but have different names:

1. IndentationError
2. TabError

These exceptions both inherit from the SyntaxError class, but they're special cases where indentation is concerned. An IndentationError is raised when the indentation levels of your code don't match up. A TabError is raised when your code uses both tabs and spaces in the same file.

Common Syntax Problems:

Misusing the Assignment Operator (=)

```
>>> len('hello') = 5
  File "<stdin>", line 1
SyntaxError: can't assign to function call

>>> 'foo' = 1
  File "<stdin>", line 1
SyntaxError: can't assign to literal

>>> 1 = 'foo'
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Misspelling, Missing, or Misusing Python Keywords

Python keywords are a set of **protected words** that have special meaning in Python. These are words you can't use as identifiers, variables, or function names in your code. They're a part of the language and can only be used in the context that Python allows.

There are three common ways that you can mistakenly use keywords:

1. **Misspelling** a keyword
2. **Missing** a keyword
3. **Misusing** a keyword

If you **misspell** a keyword in your Python code, then you'll get a `SyntaxError`. For example, here's what happens if you spell the keyword `for` incorrectly

```
>>> fro i in range(10):
  File "<stdin>", line 1
    fro i in range(10):
        ^
SyntaxError: invalid syntax
```

Another common issue with keywords is when you **miss** them altogether:

```
>>> for i range(10):
  File "<stdin>", line 1
    for i range(10):
              ^
SyntaxError: invalid syntax
```

^^

Exceptions:

What is Exception?
An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 ))
 File "<stdin>", line 1
   print( 0 / 0 ))
              ^
SyntaxError: invalid syntax
```

The arrow indicates where the parser ran into the **syntax error**. In this example, there was one bracket too many. Remove it and run your code again:

```
>>> print( 0 / 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This time, you ran into an **exception error**. This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a ZeroDivisionError. Python comes with various built-in exceptions as well as the possibility to create self-defined exceptions.

Raising an Exception

We can use **`raise`** to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

Use raise to force an exception:

you can choose to throw an exception if a condition occurs. To throw (or raise) an exception, use the raise keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1

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

If you want to throw an error when a certain condition occurs using `raise`, you could go about it like this:

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

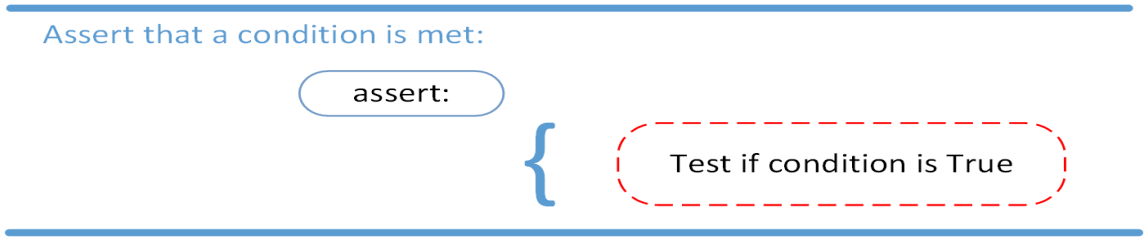
When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

The `AssertionError` Exception

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We `assert` that a certain condition is met. If this condition turns out to be `True`, then that is excellent! The program can continue. If the condition turns out to be `False`, you can have the program throw an `AssertionError` exception.



Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be `False` and the result would be the following:

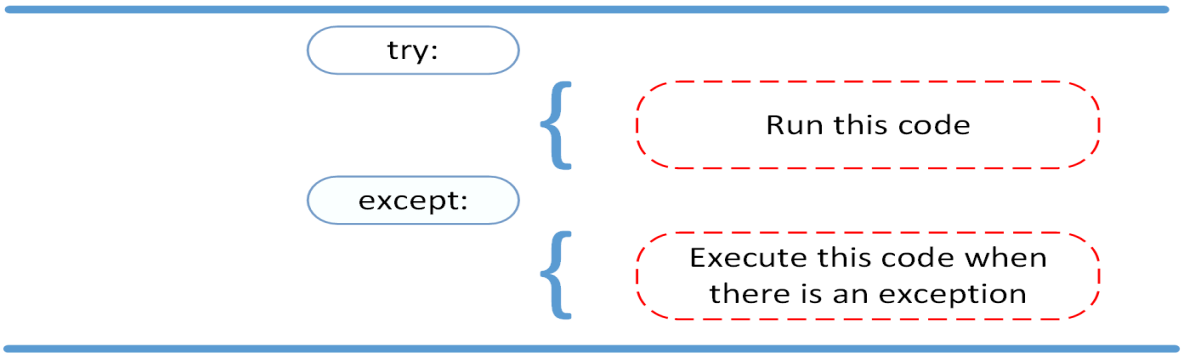
```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

In this example, throwing an `AssertionError` exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

The `try` and `except` Block: Handling Exceptions

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

The `try` and `except` block in Python is used to catch and handle exceptions. Python executes code following the `try` statement as a "normal" part of the program. The code that follows the `except` statement is the program's response to any exceptions in the preceding `try` clause.



As you saw earlier, when syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The `except` clause determines how your program responds to exceptions.

- ☐ 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.

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

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

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

```
This statement will raise an error, because x is not defined:

print(x)
```

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")
```

output:

Variable x is not defined

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The **finally:** block is a place to put

any code that must execute, whether the try-block raised an exception or not. The syntax

of the try-finally statement

```
try:
        You do your operations here;
      ......................
    Due to any exception, this may be skipped.
finally:
        This would always be executed.
      ......................
```

Example:

```
try:
        fh = open("testfile", "w")
        fh.write("This is my test file for exception handling!!")
finally:
        print ("Error: can\'t find file or read data")
        fh.close()
```

Example:

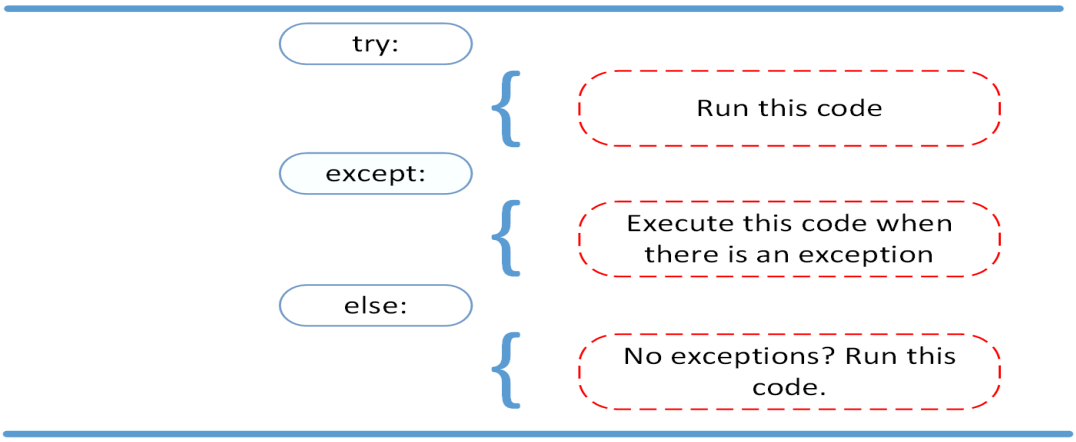
```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Else

You can use the `else` keyword to define a block of code to be executed if no errors were raised:

The else Clause

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.



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")
```

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")
```

This can be useful to close objects and clean up resources:

Example

Try to open and write to a file that is not writable:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

output:

Something went wrong when writing to the file

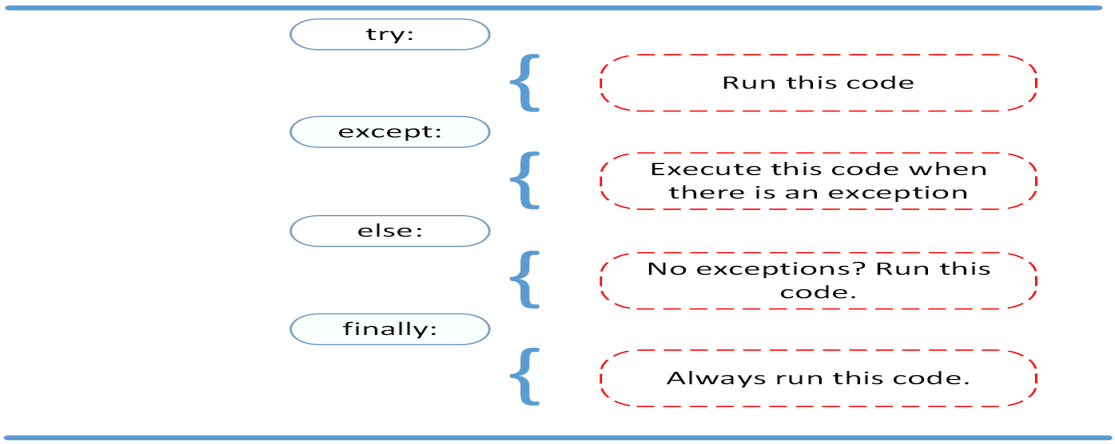
Defining Clean-up Actions (cleaning Up After Using finally):

Clean up actions are those statements within a program that are always executed. These statements are executed even if there is an error in the program. If we have used exception handling in our program then also these statements get executed.

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the finally clause.

We use try statement which has an optional clause – "finally" to perform clean up actions that must be executed under all conditions.

Cleanup actions: Before leaving the try statement, "finally" clause is always executed, whether any exception is raised or not. These are clauses which are intended to define clean-up actions that must be executed under all circumstances. Whenever an exception occurs and is not being handled by the except clause, first finally will occur and then the error is raised as default



Ex1:Code works normally and clean-up action is taken at the end

```
# Python code to illustrate  clean up actions

def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        print("I'm finally clause, always raised !! ")

# Look at parameters and note the working of Program
divide(3, 2)
```

output:

```
Yeah ! Your answer is : 1

I'm finally clause, always raised !!
```

Ex2: Code raise error and is carefully handled in the except clause. Note that Clean-up action is taken at the end.

```
# Python code to illustrate, clean up actions
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        print("I'm finally clause, always raised !! ")

# Look at parameters and note the working of Program
divide(3, 0)
Output:
```

```
Sorry ! You are dividing by zero

I'm finally clause, always raised !!
```

Ex3: raise error but we don't have any except clause to handle it. So, clean-up action is taken first and then the error(by default) is raised by the compiler.

```
# Python code to illustrate, clean up actions
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional Part as Answer
        result = x // y
    except ZeroDivisionError:
```

```
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        print("I'm finally clause, always raised !! ")

# Look at parameters and note the working of Program
divide(3, "3")
```

output:

```
I'm finally clause, always raised !!
```

Error:

```
Traceback (most recent call last):
  File "C:/Users/DELL/Desktop/Code.py", line 15, in
    divide(3, "3")
  File "C:/Users/DELL/Desktop/Code.py", line 7, in divide
    result = x // y
TypeError: unsupported operand type(s) for //: 'int' and 'str'
```

Predefined Clean-up Actions:

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
for line in open("myfile.txt"):
    print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications.

The with statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

Example:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

After the statement is executed, the file f is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

- `raise` allows you to throw an exception at any time.
- `assert` enables you to verify if a certain condition is met and throw an exception if it isn't.
- In the `try` clause, all statements are executed until an exception is encountered.
- `except` is used to catch and handle the exception(s) that are encountered in the try clause.
- `else` lets you code sections that should run only when no exceptions are encountered in the try clause.
- `finally` enables you to execute sections of code that should always run, with or without any previously encountered exceptions.

Creating User-defined Exception

Programmers may name their own exceptions by creating a new exception class. Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in **"Error"** similar to the naming of the standard exceptions in python. For example:

```
# A python program to create user-defined exception

# class MyError is derived from super class Exception
class MyError(Exception):

    # Constructor or Initializer
    def __init__(self, value):
        self.value = value

    # __str__ is to print() the value
    def __str__(self):
        return(repr(self.value))

try:
    raise(MyError(3*2))

# Value of Exception is stored in error
except MyError as error:
    print('A New Exception occured: ',error.value)
```

output:

```
('A New Exception occured: ', 6)
```

^^

Coding Simple GUI-Based Programs:

Most of the programs we have done till now are text-based programming. But many applications need GUI (**Graphical User Interface**).

Python provides several different options for writing GUI based programs. These are listed below:

- **Tkinter**: It is easiest to start with. Tkinter is Python's standard GUI (graphical user interface) package. It is the most commonly used toolkit for GUI programming in Python.
- **JPython**: It is the Python platform for Java that is providing Python scripts seamless access o Java class Libraries for the local machine.
- **wxPython**: It is an open-source, cross-platform GUI toolkit written in C++. It is one of the alternatives to Tkinter, which is bundled with Python.

There are many other interfaces available for GUI. But these are the most commonly used ones. In this, we will learn about the basic GUI programming using Tkinter.

Tkinter:

It is the standard GUI toolkit for Python. Fredrik Lundh wrote it. For modern Tk binding, Tkinter is implemented as a Python wrapper for the Tcl Interpreter embedded within the interpreter of Python. Tk provides the following widgets:

- button
- canvas
- combo-box
- frame
- level
- check-button
- entry
- level-frame
- menu
- list - box
- menu button
- message
- tk_optoinMenu

- progress-bar
- radio button
- scroll bar
- separator
- tree-view, and many more.