Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Exception Handling in Python

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

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

In [1]:
```
# Example-01
avg_sum=1
j=1
print('***** *****')
while j<5
    print('Hello world')
    j=j+1
```

```
  Input In [1]
    while j<5
             ^
SyntaxError: invalid syntax
```

In [1]:
```
# Example-02
print(10/0))
```

```
  Input In [1]
    print(10/0))
                ^
SyntaxError: unmatched ')'
```

The arrow indicates where the parser ran into the syntax error. In this example, there was one bracket too many.

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.

Errors detected during execution are called exceptions and are not unconditionally fatal: you will soon learn how to handle them in Python programs.

Most exceptions are not handled by programs, however, and result in error messages as shown here:

In [ ]:
```
print(10/0)       #ZeroDivisionError: division by zero
```

In [ ]:
```
print(4 + x * 3) #NameError: name 'x' is not defined
```

In [ ]:
```
print('5' + 5)  #TypeError: can only concatenate str (not "int") to str
```

Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError.

The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions

Note:

- A Python program terminates as soon as it encounters an error.
- In Python, an error can be
  - a syntax error or
  - an exception.
- Standard exception names are built-in identifiers (not reserved keywords).

Exception: Exceptional story

A program executes some piece of code normally. Sometimes it may take an abnormal path due to runtime errors.

An exception does not always mean it as an error.

If we walk through some iterable using the for loop, we should come to an end of the iterable at some time. Then python signals it as StopIteration.

In [7]:
```
fruits_list = [ 'apple', 'pineapple', 'fig', 'mango' ]
'''
iter_obj=iter(fruits_list)
print(next(iter_obj))    # prints first-element apple
print(next(iter_obj))    # prints second-element pineapple
print(next(iter_obj))    # prints third-element fig
print(next(iter_obj))    # prints fourth-element apple
print(next(iter_obj))    # raises StopIteration exception
'''
for fruit in fruits_list:
    print(fruit)
```

```
apple
pineapple
fig
mango
```

Sometimes exception could also be an error.

The index specified to get an element of the list may be beyond the list boundary. Then python indicates this as index error.

In [3]:
```
fruits_list = [ 'apple', 'pineapple', 'fig', 'mango' ]
for i in range(len(fruits_list)+1):
    print(fruits_list[i])       # IndexError: list index out of range
```

```
apple
pineapple
```

```
fig
mango
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [3], in <cell line: 2>()
      1 fruits_list = [ 'apple', 'pineapple', 'fig', 'mango' ]
      2 for i in range(len(fruits_list)+1):
----> 3     print(fruits_list[i])

IndexError: list index out of range
```

When an exception occurs, the program will be terminated abnormally. Can we avoid this abnormal termination? Can we get a chance to proceed further? Can we have graceful degradation?

Answer: YES, using concept called exception handling.

A few components and terminologies used with exception handling:

1. try:

The try block lets you test a block of code for errors. We indicate to the Python runtime that something unusual could happen in the suite of code.

The try statement works as follows.

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the try clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.

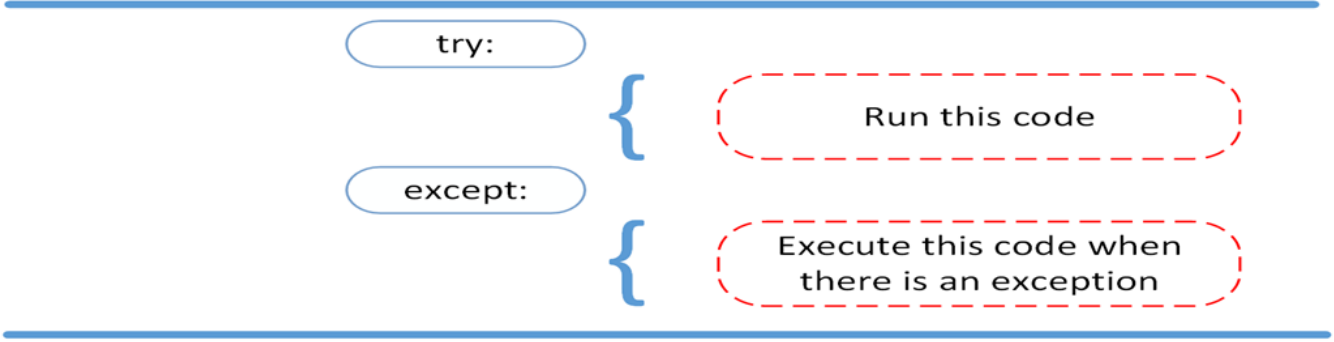
2. except [type]:

The except block lets you handle the error.

We should have at least one except block to which the control shall be transferred if and only if something unusual happens in the try suite.

So try block may be followed by one or more except block – all but one of them specifying the name of the exception – one of them may provide a common and default way of handling all exceptions.

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 block.



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 block determines how your program responds to exceptions.

Here's an example where you open a file and use a built-in exception:

In [12]:
```
with open('file2.txt') as file:
        read_data = file.read()
        print('The File contents are:')
        print(read_data)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Input In [12], in <cell line: 1>()
----> 1 with open('file2.txt') as file:
      2         read_data = file.read()
      3         print('The File contents are:')

FileNotFoundError: [Errno 2] No such file or directory: 'file2.txt'
```

In [11]:
```
try:
    with open('file2.txt') as file:
        read_data = file.read()
        print('The File contents are:')
        print(read_data)
except:      # default except block with no exception specified
    print('Could not open file2.txt')
```

```
Could not open file2.txt
```

If file2.txt does not exist, this block of code will output the following:

Could not open file2.txt

This is an informative message, and our program will still continue to run.

In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

Exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. To catch this type of exception and print it to screen, you could use the following code:

In [13]:
```
try:
    with open('file2.txt') as file:
        read_data = file.read()
        print('The File contents are:')
        print(read_data)
```

```
except FileNotFoundError:
    print('Could not open file2.txt')
```

```
Could not open file2.txt
```

In [14]:
```
try:
    with open('file2.txt') as file:
        read_data = file.read()
        print('The File contents are:')
        print(read_data)

except FileNotFoundError as fnfe:
    print(fnfe)
```

```
[Errno 2] No such file or directory: 'file2.txt'
```

https://docs.python.org/3/library/errno.html

In this case, if file2.txt does not exist, the output will be the following:

[Errno 2] No such file or directory: 'file2.txt'

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered.

In [16]:
```
def fun(a):
    if a < 4:
        b = a/(a-3)             # throws ZeroDivisionError for a = 3
    print("Value of b = ", b)   # throws NameError if a >= 4

try:
    #fun(3)
    fun(5)
    print('Last statement of try block')

except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
except:
    print("Error is caught")

print('Statement after try-except block')
```

```
NameError Occurred and Handled
Statement after try-except block
```

In [1]:
```
a = [11, 12, 13]
try:
    print ("First element = %d" %(a[0]))
    print ("Second element = %d" %(a[1]))
    print ("Third element = %d" %(a[2]))
    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except IndexError as IE:
    print(type(IE))     # the exception instance
    print(IE.args)      # arguments stored in .args
    print(IE)           # __str__ allows args to be printed directly
```

```
First element = 11
Second element = 12
Third element = 13
<class 'IndexError'>
```

```
('list index out of range',)
list index out of range
```

In [3]:
```
a = [11, 12, 13]
try:
    for i in range(5):
        print ("element %d is %d" %(i,a[i]))
        # Throws error when i=4 since there are only 3 elements in array

except IndexError as IE:
    print(type(IE))     # the exception instance
    print(IE.args)      # arguments stored in .args
    print(IE)           # __str__ allows args to be printed directly
```

```
element 0 is 11
element 1 is 12
element 2 is 13
<class 'IndexError'>
('list index out of range',)
list index out of range
```

An except clause may name multiple exceptions as a parenthesized tuple, for example:

In [4]:
```
try:
    #print(avg_result)
    #print(10/0)
    print(2+'5')
except (NameError, ZeroDivisionError, TypeError):
    print('Exception is caught')
```

```
Exception is caught
```

3. builtin exceptions:

There are a number of exceptions which Python knows. We call them built-in exceptions. Examples:

1. IndexError: Raised when an index is not found in a sequence.
2. KeyError: Raised when the specified key is not found in the dictionary.
3. NameError: Raised when an identifier is not found in the local or global namespace.
4. IndentationError: If incorrect indentation is given.
5. IOError: It occurs when Input Output operation fails.
6. EOFError: It occurs when the end of the file is reached, and yet operations are being performed.
7. RuntimeError: Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.
8. StopIteration: Raised by built-in function next() and an iterator's **next**() method to signal that there are no further items produced by the iterator.
9. TypeError: Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
10. OverflowError: Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise MemoryError than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range.
11. ImportError: Raised when the import statement has troubles trying to load a module. Also raised when the "from list" in from ... import has a name that cannot be found.
12. AssertionError: Raised when an assert statement fails.
13. AttributeError": Raised when an attribute reference (see Attribute references) or assignment fails. (When an object does not support attribute references or attribute assignments at all, TypeError is raised.)

14. ModuleNotFoundError: A subclass of ImportError which is raised by import when a module could not be located.
15. OSError([arg]): This exception is raised when a system function returns a system-related error, including I/O failures such as “file not found” or “disk full” (not for illegal argument types or other incidental errors).

These are automatically raised when they happen.

In [10]:
```
try:
    def f():
        z=['foo','bar']
        for i in z:
            if i == 'foo':
                print('foo')
except IndentationError as e:
    print(e)
```

4. raising an exception:

So far, you have only been catching exceptions that are thrown by the Python run-time system. However, it is possible for your program to throw an exception explicitly, using the raise statement. The general form of raise is shown here:

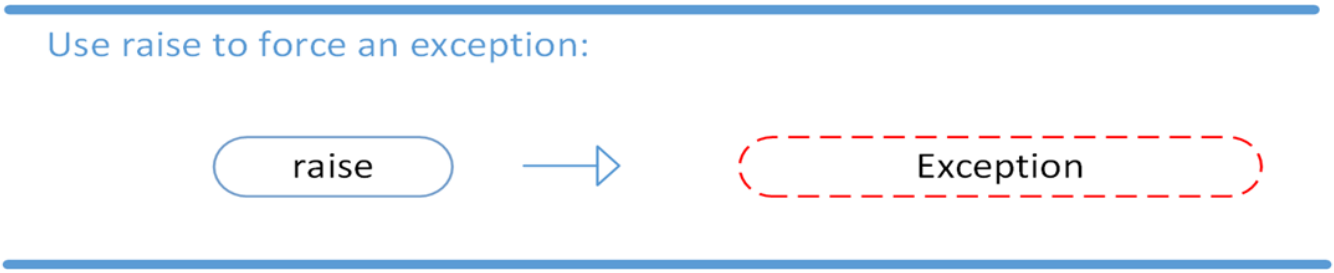
raise Exception_name()

Here, Exception_name must be a subclass of Exception.

This causes creation of an exception object with the message. The object also remembers the place in the code (line number, function name, how this function got called on). The raising of exception causes the program to abort if the raise statement is not in a try block or transfer the control to the end of try block to match the except blocks.

Raising an Exception:

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



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

In [18]:
```
# user defned exception
class AgeBarException(Exception):
    def __init__(self, str):
        self.str = str
    def __str__(self):
        return self.str

name=input('Enter name: ')
age = int(input('Enter age: '))
```

```
if age > 70:
    raise AgeBarException(f'age should not exceed 70. The value of age was: {age}')
else:
    print('Name =',name,'  Age =', age)
    print('Driving is allowed')

print('Statement after try-except block')
```

```
Enter name: raj
Enter age: 76
```

```
---------------------------------------------------------------------------
AgeBarException                           Traceback (most recent call last)
Input In [18], in <cell line: 11>()
      9 age = int(input('Enter age: '))
     11 if age > 70:
---> 12     raise AgeBarException(f'age should not exceed 70. The value of age was: {age
}')
     13 else:
     14     print('Name =',name,'  Age =', age)

AgeBarException: age should not exceed 70. The value of age was: 76
```

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

In [8]:
```
name=input('Enter name: ')
age = int(input('Enter age: '))
try:
    if age > 70:
        raise AgeBarException(f'age should not exceed 70. The value of age was: {age}')
    else:
        print('Name =',name,'  Age =', age)
        print('Driving is allowed')
except AgeBarException as e:
    print(type(e))     # the exception instance
    print(e.args)      # arguments stored in .args
    print(e)

print('Statement after try-except block')
```

```
Enter name: raj
Enter age: 80
<class '__main__.AgeBarException'>
('age should not exceed 70. The value of age was: 80',)
age should not exceed 70. The value of age was: 80
Statement after try-except block
```

In [21]:
```
try:
    raise Exception('spam', 'eggs')
except Exception as e:
    print(type(e))     # the exception instance
    print(e.args)      # arguments stored in .args
    print(e)           # __str__ allows args to be printed directly,
                           # but may be overridden in exception subclasses
    x, y = e.args      # unpack args
    print('x =', x)
    print('y =', y)
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

5. matching of except blocks:

- The raised or thrown exception object is matched with the except blocks in the order in which they occur in the try-except statement.
- The code following the first match is executed. It is always the first match and not the best match.
- If no match occurs and there is a default except block (with no exception specified), then that block will be executed.

Example 1:

A class in an except clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class).

For example, the following code will print B, C, D in that order:

In [15]:
```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    #except Exception:
    #    print("*")
    except D:
        print(D)
    except C:
        print(C)
    except B:
        print(B)
```

```
<class '__main__.B'>
<class '__main__.C'>
<class '__main__.D'>
```

Note that the following code will print B, B, B — the first matching except clause is triggered.

In [18]:
```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except B:
        print("B")
    except C:
        print("C")
    except D:
        print("D")
```

```
B
B
B
```

Example 2:

In the below code, the last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

In [14]:
```
import sys    # System-specific parameters and functions

try:
    f = open('t.txt')  # myfile.txt
    s = f.readline()
    i = int(s.strip())

except OSError as err:    #FileNotFoundError is a subclass of OSError
    print("OS error: {0}".format(err))

except ValueError:
    print("Could not convert data to an integer.")

except:
    print("Unexpected error:", sys.exc_info()[0])
    raise

"""The sys module provides one function that provides the details of
the exception that was raised. Programs with exception handling will
occasionally use this function.

The sys.exc_info function returns a 3-tuple with
    the exception,
    the exception's parameter, and
    a traceback object that pinpoints the line of Python that raised the exception.
"""
```

Out[14]:
```
Could not convert data to an integer.
"The sys module provides one function that provides the details of \nthe exception that
was raised. Programs with exception handling will \noccasionally use this function.\n\nT
he sys.exc_info function returns a 3-tuple with \n    the exception, \n    the exceptio
n's parameter, and \n    a traceback object that pinpoints the line of Python that raise
d the exception.\n"
```

Note:

errno : A numeric error code from the C variable errno. https://docs.python.org/3/library/errno.html

6. finally block:

This is optional. This follows all the except blocks. It is part of the try statement. This block shall be executed on both normal flow and exceptional flow.

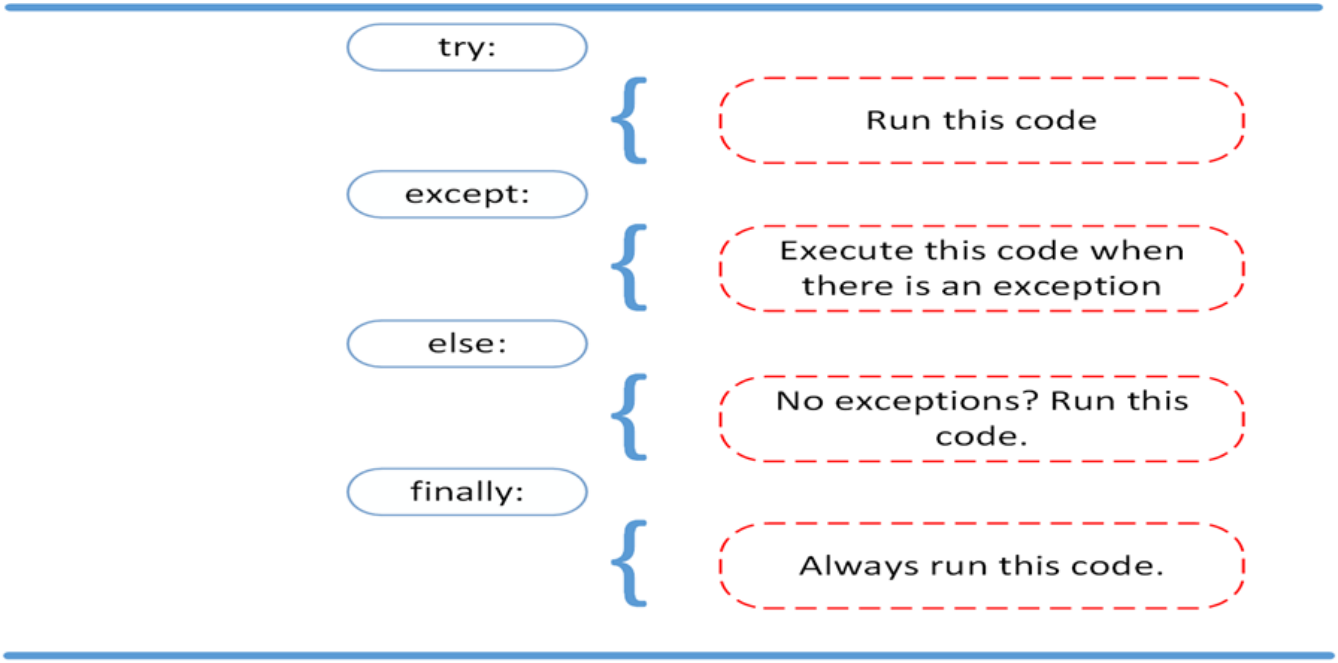
Let us understand the flow of execution.

- Normal flow:
  - try block – finally block if any – code following try block
- Exceptional flow
  - try block – exit the try block on an exception – find the first matching except block – execute the matched except block – finally block – code following try block.

Observe there is no mechanism in any language including Python to go back to the try block – no way to resume at the point of exception.

Cleaning up after using by finally block

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.



Have a look at the following example:

In [20]:
```
import sys
'''
The sys module in Python provides various functions and variables that are
used to manipulate different parts of the Python runtime environment.
'''
def linux_interaction():
    #print(sys.version)  #prints a string containing the version of Python Interpreter w
    #print(sys.platform)
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')

try:
    linux_interaction()  # this function can only run on Linux systems
except AssertionError as AE:
    print(AE)
else:
    try:
        with open('file1.txt') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

```
Function can only run on Linux systems.
Cleaning up, irrespective of any exceptions.
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

Function can only run on Linux systems. Cleaning up, irrespective of any exceptions.

sys — System-specific parameters and functions

sys module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

The possible return values from the following command.

import sys

print(sys.platform)

System	Value
Linux	linux or linux2 (*)
Windows	win32
Windows/Cygwin	cygwin
Windows/MSYS2	msys
Mac OS X	darwin
OS/2	os2
OS/2 EMX	os2emx
RiscOS	riscos
AtheOS	atheos
FreeBSD 7	freebsd7
FreeBSD 8	freebsd8
FreeBSD N	freebsdN
OpenBSD 6	openbsd6

(*) Prior to Python 3.3, the value for any Linux version is always `linux2`; after, it is `linux`.

7. user defined exception:

It is not possible for a language to specify all possible unusual cases. So the users can also specify exception as a class which inherits from a class called Exception.

Often you'll need to throw something other than one of the existing exceptions; they aren't always the best choice. Let's take a look at creating your own types of exceptions. You do this by subclassing one of the existing Exception subclasses.

The following code is from the file 4_exception.py. It shows how to make our own exception object.

In [23]:
```
# user defned exception
class MyException(Exception):
    def __init__(self, str):
        self.str = str
    def __str__(self):
        return self.str

# check whether n is between 1 and 100
n = int(input("Enter a number:"))
try:
    if not 1 <= n <= 100 :
        # Creates and throws MyException object
        raise MyException("Number not in range")
    print("Number is fine : ", n)
```

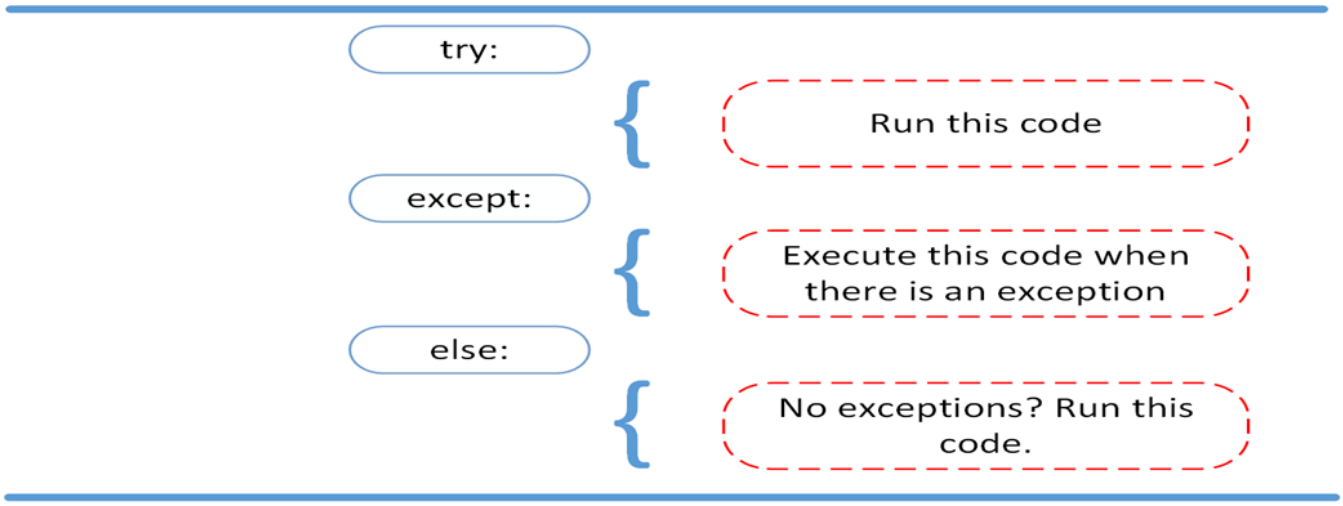
```
except MyException as e:
    print(e)        # calls  MyException.__str__(e)

print("Thats all")
```

```
Enter a number:500
Number not in range
Thats all
```

8. 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.



Look at the following example:

In [25]:
```
import sys

def linux_interaction():
    #assert ('win32' in sys.platform)
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
    print('Doing something.')

try:
    linux_interaction()
except AssertionError as error:
    print(error)
else:
    print('Executing the else clause.')
```

```
Doing something.
Executing the else clause.
```

9. exception propagation:

We say that a try block has dynamic scope. Any exception raised in the block or any function called from there are considered part of the try block. Any exception raised within these functions called from there will cause the control to be propagated backwards to the except blocks of this try block. Let us go through a few examples.

The following code is taken from the file 2_exception_intro.py. You will observe that the program aborts as soon as any one of these statements is executed. You get know a few builtin exceptions in this program.

In [ ]:
```
# example 1
# print("res : ", 10 / 0)  # ZeroDivisionError: division by zero

# example 2
# print(myvar)                    # NameError: name 'myvar' is not defned

# example 3
# open("unknown.txt")  #FileNotFoundError: [Errno 2] No such fle or directory:'unknown.t
```

Let us observe some code from the file 2_exception.py

In [28]:
```
m = 10
#n = 2
n = 0
try:
    print("one")
    print("res : ", m / n)
    print("two")
    print("three")
except Exception as e:
    print(e)

print("four")
```

```
one
division by zero
four
```

In the above code, print("two") statement is executed only if the earliest statement does not raise an exception.

On an exception, object e of the class DivisionByException is created. This class is the derived class of class Exception. The variable is of type Exception. An object of base class can always receive an object of the derived class. print(e) calls e.**str**() and displays the resulting string.

In [30]:
```
m = 10
#n = 2
n = 0
try:
    print("one")
    #print("res : ", m / n)
    print("two")
    print(myvar)
    print("three")

except NameError as e:
    print("no such name : ", e)

#except Exception as e:
# print("all other exceptions : ", e)

except:
    print("all exceptions")
```

```
one
two
no such name :  name 'myvar' is not defined
```

Also observe that on an exception:

- first match is executed
- one and only one except block is executed

- there is no way to go back to the try block i.e., no resume at the point of exception.

Here are the key takeaways:

- A try clause is executed up until the point where the first exception is encountered.
- Inside the except clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.
- Avoid using bare except clauses.

Summing Up

After seeing the difference between syntax errors and exceptions, you learned about various ways to raise, catch, and handle exceptions in Python. In this article, you saw the following options:

- 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.

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
```

```
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
```

```
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- EncodingWarning
           +-- ResourceWarning
```

References:

1. 18_comprehension_exception.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://realpython.com/python-exceptions/
3. https://docs.python.org/
4. https://docs.python.org/3/library/exceptions.html