ERRORS AND EXCEPTIONS

M.Malarmathi

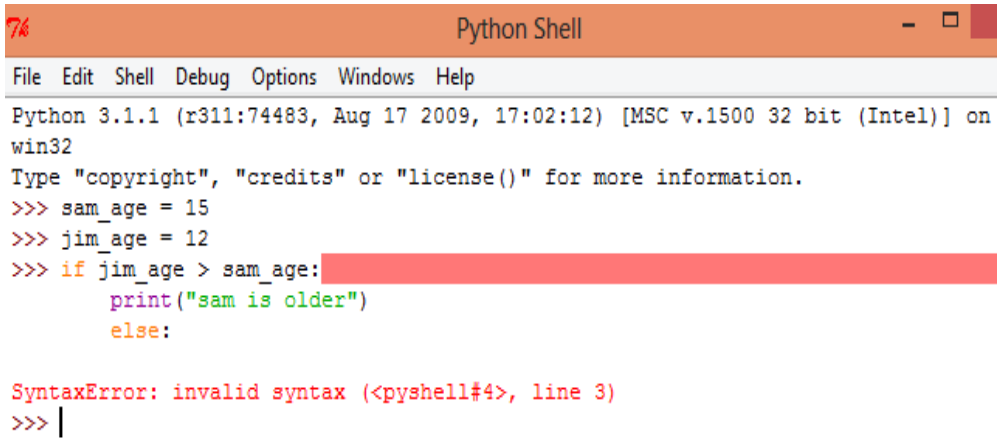
AP/IT

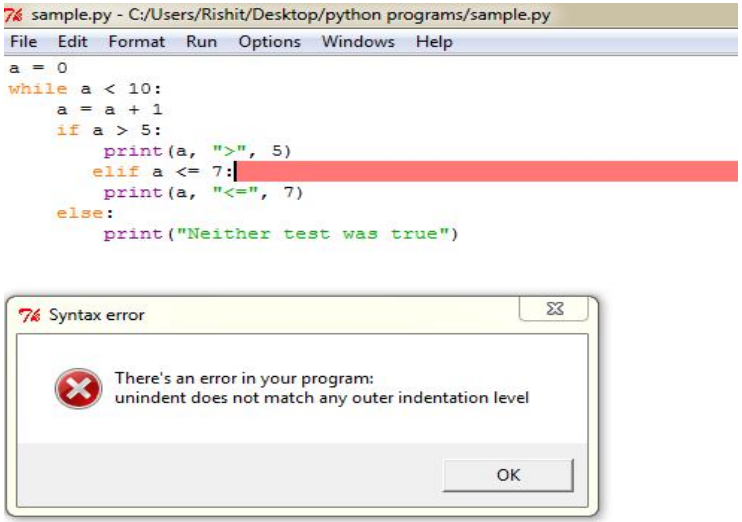
SYNTAX ERRORS

- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message.
- **Syntax** refers to the structure of a program and the rules about that structure.

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                     ^
SyntaxError: invalid syntax
```

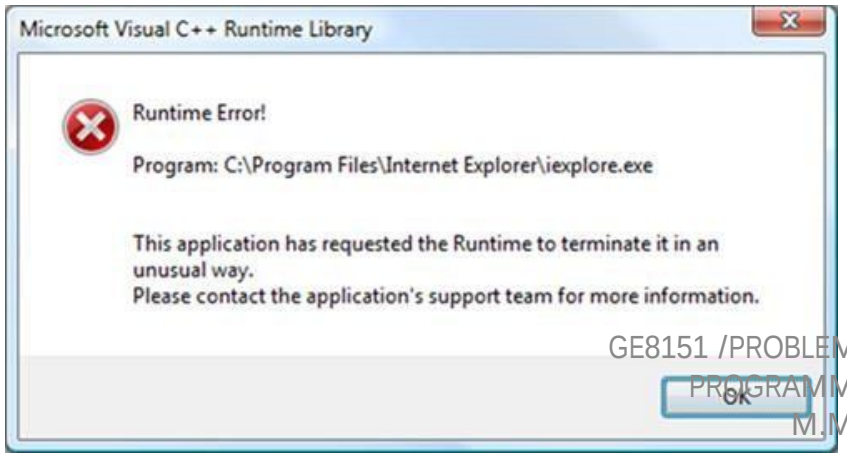
- If there is a single syntax error anywhere in your program, Python will display an error message and quit. You will not be able to complete the execution of your program.

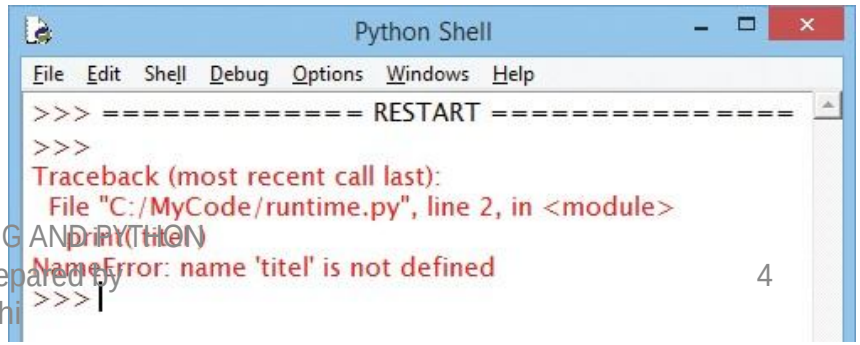




RUN TIME ERRORS

- The second type of error is a runtime error
- error does not appear until you run the program.
- These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.





SEMANTIC ERROR

- If there is a semantic error in your program, the computer will not generate any error messages.
- program will not do the right thing. It will do something else.
- The meaning of the program (its semantics) is wrong.
- Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

- Here the program is for sum of n natural numbers – but output is wrong

List of few standard Error Exceptions

IndexError	sequence subscript out of range
KeyError	dictionary key not found
ZeroDivision Error	division by 0
ValueError	value is inappropiate for the built-in function
TypeError	function or operation using the wrong type
IOError	input/output error, file handling

Exceptions

- **Exceptions** are errors that occur at the runtime of the program.
- If exceptions are not handled by the program they may result in crashing of the program (or)
- **Exceptions are** events that can modify the flow or control through a program.
- They are automatically triggered on errors.
- **try/except** : catch and recover from raised by you or Python exceptions
- **try/finally**: perform cleanup actions whether exceptions occur or not
- **raise**: trigger an exception manually in your code
- **assert**: conditionally trigger an exception in your code

Exception Roles

- **Error handling**
  - Wherever Python detects an error it raises exceptions
  - Default behavior: stops program.
  - Otherwise, code try to catch and recover from the exception (try handler)
- **Event notification**
  - Can signal a valid condition (for example, in search)
- **Special-case handling**
  - Handles unusual situations
- **Termination actions**
  - Guarantees the required closing-time operators (try/finally)
- **Unusual control-flows**
  - A sort of high-level “goto”

try/except/else

```
try:
    <block of statements>      #main code to run
except <name1>:                #handler for exception
    <block of statements>
except <name2>,<data>:         #handler for exception
    <block of statements>
except (<name3>,<name4>): #handler for exception
    <block of statements>
except:                        #handler for exception
     <block of statements>
else:              # optional, runs if no exception occurs
      <block of statements>
```

Example

```
>>>try:
      action()
    except NameError(): ...
    except IndexError():  ...
    except KeyError(): ...
    except (AttributeError,TypeError,SyntaxError):...
    else: ....
```

- General catch-all clause: add empty except.
- It may catch unrelated to your code system exceptions.
- It may catch exceptions meant for other handler (system exit)

Zero Division Example

```
>>> print 3/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> def division(x,y):
...   try:
...     return x/y
...   except ZeroDivisionError:
...     print "division by zero"
...
>>> division(5,2)
2
>>> division(5,0)
division by zero
>>>
```

```
>>> def div(x,y):
...     try:
...         if y==0: raise 'zero'
...         return x/y
...     except 'zero': print "ZERO"
...
>>> div(3,4)
0
>>> div(3,0)
ZERO
>>>
```

Value error Example

```
try:
  number = int(input("Enter a number, but not 10: "))
  if number == 10:
    raise ValueError("Oh no, not 10")
except ValueError as error:
  print("The exception is:", str(error))
else:
  print("Good job")
print("Business as usual, this will be executed.")
```

```
Enter a number, but not 10: 5
Good job
Business as usual, this will be executed.
```

```
Enter a number, but not 10: 10
The exception is: Oh no, not 10
Business as usual, this will be executed.
```

I/O exception example

How to catch file opening errors

```
import sys
try:
    infile = open('myfile.txt', 'r')
except IOError as error:
    print("Can't open file, reason:", str(error))
    sys.exit(1)
for line in infile:
    print(line, end='')
infile.close
```

```
Python is a great language.
Yeah its great!!
```

```
Can't open file, reason: [Errno 2] No such file or directory: 'hi.txt'
Traceback (most recent call last):
  File "C:/Python30/wr.py", line 6, in <module>
    sys.exit(1)
SystemExit: 1
```

Keyboard input exception example (Value error)

How to catch bad input from keyboard

```
import sys
try:
    my_number = int(input(”Please, input an integer”))
except ValueError:
    print(”You did not enter an integer”)
    sys.exit(1)
print(”The number was”, my_number)
```

Scope of my_number……

What happens if there is no sys.exit ?

try/else

- **else** is used to verify if no exception occurred in **try**.
- You can always eliminate *else* by moving its logic at the end of the *try* block.
- However, if “else statement” triggers exceptions, it would be misclassified as exception in try block.

try/finally

- In **try/finally**, *finally* block is always run whether an exception occurs or not

```
try:
   <block of statements>
finally:
   <block of statements>
```

- Ensure some actions to be done in any case
- It can not be used in the *try* with *except* and *else*.

Examples

```
>>> try:
...    print 3/0
... finally: print "finish"
...
finish
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
ZeroDivisionError: integer division or modulo by zero
>>> try:
...    try:
...     print 3/0
...    finally: print "Finish"
... except: print "Catch exception"
...
Finish
Catch exception
>>>
```

```
>>> try:
...    file=open('data','r')
... finally: file.close()
...
Traceback (most recent call last):
  File "<stdin>", line 3, in ?
TypeError: descriptor 'close' of 'file' object needs an argument
>>>
```

raise

- **raise** triggers exceptions explicitly

raise <name>
raise <name>,<data> # provide data to handler
raise #re-raise last exception

```
>>>try:
      raise ‘zero’, (3,0)
   except ‘zero’: print “zero argument”
   except ‘zero’, data: print data
```

- Last form may be useful if you want to propagate cought exception to another handler.
- Exception name: built-in name, string, user-defined class

Example

```
>>> try:
...     raise KeyboardInterrupt
... except:
...     print "propogate"
...     raise
...
propogate
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
>>>
```

Exception Objects

- **String-based exceptions** are any string object

```
>>> myException="I can make exceptions!"
>>> try:
...   raise myException
... except myException:
...   print 'caught'
...
caught
>>> raise myException
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
I can make exceptions!
>>>
```

- **Class-based exceptions** are identified with classes. They also identify categories of exceptions.
- String exceptions are matched by object identity: *is*
- Class exceptions are matched by superclass identity: *except* catches instances of the mentioned class and instances of all its subclasses lower in the class tree.

Raising an exception

It can be useful to raise exceptions yourself. You can create code that checks for errors that are not programatically wrong, but are still errors in your program.

You don't need **try** to raise an exception, but then the top level exception handler will handle it.

```
try:
    number = int(input("Enter a number, but not 10: "))
    if number == 10:
        raise ValueError("Oh no, not 10")
except ValueError as error:
    print("The exception is:", str(error))
else:
   print("Good job")
print("Business as usual, this will be executed.")
```

assert

- **assert** is a conditional *raise*

assert <test>, <data>

assert <test>

- If <test> evaluates to false, Python raises AssertionError with the <data> as the exception's extra data.

```
>>> def f(x,y):
...     assert x>0,'x must be positive'
...     assert y<0,'y must be negative'
...     return y**x
...
>>> f(4,-3)
81
>>> f(-3,-4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in f
AssertionError: x must be positive
>>> f(4,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in f
AssertionError: y must be negative
>>>
```

Assertions

Assertions are for debugging – not errors.

An assertion is conditionally rasing an exception; AssertionError.

You can choose to catch it or not, inside a **try** or not.

```
import sys
try:
    number = int(input(”Please input a small positive number:”))
    assert number > 0 and number < 10, ”Number out of range”
except ValueError:
    print(”You don’t know what a number is”)
    sys.exit(1)
except AssertionError as err:
    print(str(err))
```

You can ignore assertions by running Python with –O option.

Thank You