Errors

Syntax errors:

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

In [1]:
```
for i in range(10)
print(i)
```

```
  File "<ipython-input-1-22b14165345a>", line 1
    for i in range(10)
                      ^
SyntaxError: invalid syntax
```

In [2]:
```
print "python"
```

```
  File "<ipython-input-2-3935e50686e8>", line 1
    print "python"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("python")?
```

Exceptions

- Unwanteed event that disrupts normal flow of execution
- Errors detected during execution are called exceptions

Example: open db connection, read data, modify it, close connection
read file from remote location, if not availabe then read from local drive

In [3]:
```
1 5 * (1/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-b2666f79bf3e> in <module>
----> 1 5 * (1/0)

ZeroDivisionError: division by zero
```

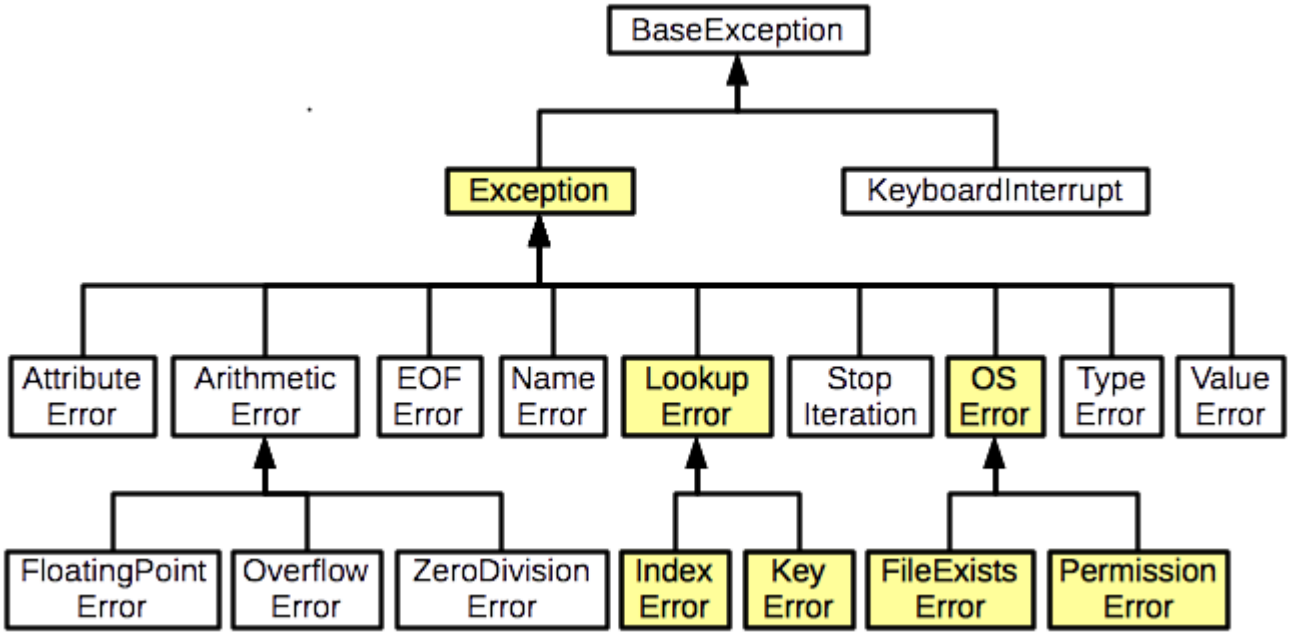
In [4]:
```
1 a=input("Enter a number: ")
2 print(a+10)
```

```
Enter a number: 10

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-85827423ef95> in <module>
      1 a=input("Enter a number: ")
----> 2 print(a+10)

TypeError: can only concatenate str (not "int") to str
```

In [5]:
```
1 f = open("output.txt")
2 f.read()
3 f.close()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-5-5bc89c2c4578> in <module>
----> 1 f = open("output.txt")
      2 f.read()
      3 f.close()

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

Exception hierarchy



Default Exception Handling

1. PVM creates object of ZeroDivision class
2. If there is not handling code then it is by default handled by PVM and rest program won't be executed
3. else, it is been handled by the handler

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

- 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.

Handling exception using try except

- put risky code inside try block
- write alternate solution in except block

In [6]:
```
1 print('Welcome')
2 a = 10/0
3 print('Hello')
4 print('Hi')
```

```
Welcome

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-5739b185508d> in <module>
      1 print('Welcome')
----> 2 a = 10/0
      3 print('Hello')
      4 print('Hi')

ZeroDivisionError: division by zero
```

In [7]:
```
1  print('Welcome')
2
3  try:
4      a = 10/0
5  except ZeroDivisionError:
6      a = 10/2
7
8  print(a)
9  print('Hello')
10 print('Hi')
```

```
Welcome
5.0
Hello
Hi
```

In [8]:
```
def m1():
    m2()
    print("hi")

def m2():
    try:
        m3()

# print Exception Information to the Console using as "e"
    except ZeroDivisionError as e:
        print("handling the code")
        print(type(e), e)

    print("bye")

def m3():
    print(10/0)
```

In [9]:
```
m1()
```

```
handling the code
<class 'ZeroDivisionError'> division by zero
bye
hi
```

In [10]:
```
# Statement raising an expection
while True:
    x = int(input("Please enter a number: "))
```

```
Please enter a number: 10
Please enter a number: 10
Please enter a number: hello

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-c4a1058b6df2> in <module>
      1 # Statement raising an expection
      2 while True:
----> 3     x = int(input("Please enter a number: "))

ValueError: invalid literal for int() with base 10: 'hello'
```

In [1]:
```
# Handling expection
while True:
    try:
        x = int(input("Please enter a number: "))
#         break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
        break
```

```
Please enter a number: 10
Please enter a number: 15
Please enter a number: 16
Please enter a number: hello
Oops!  That was no valid number.  Try again...
```

try with multiple blocks

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.

In [2]:
```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
        print("result is", y/x)

    except TypeError:
        print("convert str to int!")
        print("result is", int(x)/int(y))
```

In [3]:
```
divide(20, "10")
```

```
convert str to int!
result is 2.0
```

In [4]:
```
divide(10,0)
```

```
division by zero!
result is 0.0
```

In [5]:
```
# Super class is written followed by subclass
try:
    a=10/0
except ArithmeticError:
    print("haha")
except ZeroDivisionError:
    print("sad")
```

```
haha
```

In [6]:
```
try:
    a=10/'a'
except ZeroDivisionError:
    print("sad")
except ArithmeticError:
    print("haha")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-b0a454d18a88> in <module>
      1 try:
----> 2     a=10/'a'
      3 except ZeroDivisionError:
      4     print("sad")
      5 except ArithmeticError:

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

try with except for multiple statement--

if exception handling code is same for multpile exception, use single except block () is mandatory

In [7]:
```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
        print("result is", y/x)
    except TypeError:
        print("convert str to int!")
        print("result is", int(x)/int(y))
```

In [8]:
```
divide(3,0)
```

```
division by zero!
result is 0.0
```

In [9]:
```
1 divide('3','1')
```

```
convert str to int!
result is 3.0
```

In [10]:
```
1 def divide(x, y):
2     try:
3         result = x / y
4     except (ZeroDivisionError, TypeError) as msg:
5         print(msg)
```

In [11]:
```
1 divide(0, "0")
```

```
unsupported operand type(s) for /: 'int' and 'str'
```

In [12]:
```
1 divide(10,0)
```

```
division by zero
```

default exception block

used to handle any type of exception

In [13]:
```
1 def divide(x, y):
2     try:
3         result = x / y
4     except ZeroDivisionError:
5         print(10/2)
6     except:
7         print("hi")
```

In [14]:
```
1 divide(10, "10")
```

```
hi
```

In [15]:
```
# try reversing the order
def divide(x, y):
    try:
        result = x / y
    except:
        print("hi")
    except ZeroDivisionError:
        print(10/2)
```

```
  File "<ipython-input-15-b13418c5b1a9>", line 4
    result = x / y
    ^
SyntaxError: default 'except:' must be last
```

In [16]:
```
try:
    print(10/0)
except Exception:
    print(10/4)
except ZeroDivisionError:
    print(10/2)
except TypeError:
    print('TypeError')
```

```
2.5
```

In [17]:
```
try:
    print(10/0)
except:
    print(10/4)
except (abc, ZeroDivisionError) as msg:
    print(10/2)
```

```
  File "<ipython-input-17-0e96a66a6ae7>", line 2
    print(10/0)
    ^
SyntaxError: default 'except:' must be last
```

except ZeroDivisionError:

except (ZeroDivisionError):

except ZeroDivisionError as msg:

except (ZeroDivisionError) as msg:

except (ZeroDivisionError as msg): //invalid

except (ZeroDivisionError, TypeError):

except (ZeroDivisionError, TypeError) as msg:

except ZeroDivisionError, TypeError as msg: //invalid

except:

finally block

excecuted always irrespective of whether exception is raised/handled or not
meant for cleanup code
finally won't excutes when we are using os._exit(0) >> PVM shutdown

In [18]:
```
# if no exception
try:
    print("try")
except:
    print("except")
finally:
    print("finally")
```

```
try
finally
```

In [19]:
```
# if exception raised and handled
try:
    print("try")
    print(20/0)
except ZeroDivisionError:
    print("except")
finally:
    print("finally")
```

```
try
except
finally
```

In [20]:
```
# if exception raised and but not handled
try:
    print("try")
    print(20/0)
except TypeError:
    print("except")
finally:
    print("finally")
```

```
try
finally

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-20-4f431548b0ca> in <module>
      2 try:
      3     print("try")
----> 4     print(20/0)
      5 except TypeError:
      6     print("except")

ZeroDivisionError: division by zero
```

In [ ]:
```
# Does finally always execute?
import os

try:
    print("try")
    os._exit(0)
except TypeError:
    print("except")
finally:
    print("finally")
```

In [1]:
```
# Nested try block
try:
    print("a")
    print("b")
    try:
        print("c")
        print(10/"10")
        print("e")
    except TypeError:
        print("handled")
    finally:
        print("f")

except ZeroDivisionError as msg:
    print(msg)
finally:
    print("finally block outside")
```

```
a
b
c
handled
f
finally block outside
```

else

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

- else with for: when no break is present
- else with try: when no exception raised in try block
- can't execute both exception and else simultaneously
- else can't exists without except

In [2]:
```
for i in range(10):
    print(i)
    if(i>5):
        print('Hi')
        break
else:
    print("if no break identified")
```

```
0
1
2
3
4
5
6
Hi
```

In [3]:
```
try:
    print("try")
    print(20/0)
except:
    print("except")
else:
    print("else")
finally:
    print("finally")
```

```
try
except
finally
```

In [4]:
```
try:
    print("try")
    print(20/5)
except:
    print("except")
else:
    print("else")
finally:
    print("finally")
```

```
try
4.0
else
finally
```

In [5]:
```
f= None
try:
    f= open('D://output.txt', 'r')
except Exception as e:
    print(e)
    print("Some problem with file opening")
finally:
    if f is not None:
        f.close()
    print('file closed')
```

```
[Errno 2] No such file or directory: 'D://output.txt'
Some problem with file opening
file closed
```

In [6]:
```
try:
    print("hi")
finally:
    print("hi")
```

```
hi
hi
```

In [7]:
```
try:
    print("Try: hi")
except:
    print("Except: hi")
finally:
    print("Finally: hi")
```

```
Try: hi
Finally: hi
```

In [10]:
```
try:
    print("hi")
else:
    print("hi")
except:
    print("hi")
```

```
  File "<ipython-input-10-58f7cda27fb5>", line 3
    else:
    ^
SyntaxError: invalid syntax
```

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.

In [11]:
```
1 x = -1
2
3 if x < 0:
4     raise Exception("Sorry, no numbers below zero")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-11-38174769ef86> in <module>
      2
      3 if x < 0:
----> 4     raise Exception("Sorry, no numbers below zero")

Exception: Sorry, no numbers below zero
```

In [12]:
```
1 x = "hello"
2
3 if not type(x) is int:
4     raise TypeError("Only integers are allowed")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-75047f5dae4a> in <module>
      2
      3 if not type(x) is int:
----> 4     raise TypeError("Only integers are allowed")

TypeError: Only integers are allowed
```

In [13]:
```
try:
    x = input('Enter Name: ')
    if x == 'Hello':
        raise Exception('Name cannot be Hello')
except:
    print("Exception Raised. Default name is ABC")
    x = 'ABC'
finally:
    print('Name is: ' + x)
```

```
Enter Name: Hello
Exception Raised. Default name is ABC
Name is: ABC
```

assert keyword

Assertion is a programming concept used while writing a code where the user declares a condition to be true using assert statement prior to running the module. If the condition is True, the control simply moves to the next line of code. In case if it is False the program stops running and returns AssertionError Exception.

Syntax of assertion: assert condition, error_message(optional)

In [14]:
```
x = 1
y = 0
assert y != 0, "Invalid Operation"          # denominator can't be 0
print(x / y)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-14-c9cd278718e6> in <module>
      1 x = 1
      2 y = 0
----> 3 assert y != 0, "Invalid Operation"          # denominator can't be 0
      4 print(x / y)

AssertionError: Invalid Operation
```

Handling AssertionError exception: AssertionError is inherited from Exception class, when this exception occurs and raises AssertionError there are two ways to handle, either the user handles it or the default exception handler.

The default exception handler in python will print the error_message written by the programmer, or else will just handle the error without any message. We have seen this in the above example.

In [15]:
```
# Handling exception manually:
try:
    x = 1
    y = 0
    assert y != 0, "Invalid Operation"
    print(x / y)

# the errror_message provided by the user gets printed
except AssertionError as msg:
    print(msg)
```

```
Invalid Operation
```

In [17]:
```
try:
    x = input('Enter Name: ')
    assert x!= 'Hello', 'Name cannot be Hello'
except AssertionError:
    print("AssertionError Exception Raised.")
    x = 'ABC'
finally:
    print('Name is: ' + x)
```

```
Enter Name: Hello
AssertionError Exception Raised.
Name is: ABC
```

In [ ]:
```

```