HAB 619 Introduction to Scientific Computing in Sports Science

De Motu Animalium G.Borelli (1680)

#6

SERDAR ARITAN

serdar.aritan@hacettepe.edu.tr

Biyomekanik Araştırma Grubu
www.biomech.hacettepe.edu.tr
Spor Bilimleri Fakültesi
www.sbt.hacettepe.edu.tr
Hacettepe Universitesi, Ankara, Türkiye
www.hacettepe.edu.tr

General philosophy of errors and exception handling

Any program may encounter errors during its execution. For the purposes of illustrating exceptions, we'll look at the case of a word processor that writes files to disk and that therefore may run out of disk space before all of its data is written. There are various ways of coming to grips with this problem.

SOLUTION 1: DON'T HANDLE THE PROBLEM

The simplest way of handling this disk-space problem is to assume that there will always be adequate disk space for whatever files we write, and we needn't worry about it. Unfortunately, this seems to be the most commonly used option. It's usually tolerable for small programs dealing with small amounts of data, but it's completely unsatisfactory for more mission-critical programs.

General philosophy of errors and exception handling

SOLUTION 2: ALL FUNCTIONS RETURN SUCCESS/FAILURE STATUS

The next level of sophistication in error handling is to realize that errors will occur and to define a **methodology** using standard language mechanisms for detecting and handling them. There are various ways of doing this, but a typical one is to have **each function** or procedure return **a status value** that indicates if that function or procedure call executed successfully. Normal results can be passed back in a call-by-reference parameter.

General philosophy of errors and exception handling

SOLUTION 3: THE EXCEPTION MECHANISM

It's obvious that most of the error-checking code in the previous type of program is largely repetitive: it checks for errors on each attempted file write and passes an error status message back up to the calling procedure if an error is detected. The disk space error is handled in only one place, the top-level save_to_file.

```
function save_to_file(filename)
        try to execute the following block
                save_text_to_file(filename)
                save_formats_to_file(filename)
                save_prefs_to_file(filename)
....
        except that, if the disk runs out of space while
        executing the above block, do this
        ...handle the error...
```

General philosophy of errors and exception handling

The act of generating an exception is called raising or throwing an exception. The act of responding to an exception is called catching an exception, and the code that handles an exception is called exception-handling code, or just an exception handler. Depending on exactly what event causes an exception, a program may need to take different actions. For example, an exception raised when disk space is exhausted needs to be handled quite differently from an exception that is raised if we run out of memory, and both are completely different from an exception that arises when a divide-by-zero error occurs.

General philosophy of errors and exception handling

Like everything else in Python, an exception is an object. It's generated automatically by Python functions with a raise statement. After it's generated, the raise statement, which **raises an exception, causes execution of the Python program to proceed in a manner different than would normally occur**. Instead of proceeding with the next statement after the raise, or whatever generated the exception, the current call chain is searched for a handler that can handle the generated exception. If such a handler is found, it's invoked and may access the exception object for more information. If no suitable exception handler is found, the program aborts with an error message.

General philosophy of errors and exception handling

Types of Python exceptions: It's possible to generate different types of exceptions to reflect the actual cause of the error or exceptional circumstance being reported. Python provides a number of different exception types:

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
    StopIteration
    ArithmeticError
      FloatingPointError
      OverflowError
      ZeroDivisionError
  AssertionError
  AttributeError
  BufferError
EnvironmentError
  IOError
  OSError
   WindowsError (Windows)
   VMSError (VMS)
EOFError
ImportError
LookupError
  IndexError
  KeyError
MemoryError
NameError
  UnboundLocalError
ReferenceError
RuntimeError
  NotImplementedError
SyntaxError
  IndentationError
    TabError
SystemError
TypeError
ValueError
  UnicodeError
    UnicodeDecodeError
    UnicodeEncodeError
    UnicodeTranslateError
```

Exception Handling

Different exception types

Console 1/A

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.21.0 -- An enhanced Interactive Python.

In [1]: 10*(1/0)
Traceback (most recent call last):

  File "<ipython-input-1-fbc7e86ae71a>", line 1, in <module>
    10*(1/0)

ZeroDivisionError: division by zero


In [2]: 4+spam*3
Traceback (most recent call last):

  File "<ipython-input-2-31e460ab435d>", line 1, in <module>
    4+spam*3

NameError: name 'spam' is not defined


In [3]: '2' + 2
Traceback (most recent call last):

  File "<ipython-input-3-d2b23a1db757>", line 1, in <module>
    '2' + 2

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


In [4]:
```

IPython console | History

Exception Handling

Value Error : input only number example

Console 1/A

```
In [6]: while True:
   ...:     try:
   ...:         x = int(input("Please enter a number : "))
   ...:         break
   ...:     except ValueError:
   ...:         print("opps! That was not a valid number. Try Again...")
   ...:

Please enter a number : serdar
opps! That was not a valid number. Try Again...

Please enter a number : 5s
opps! That was not a valid number. Try Again...

Please enter a number : 67-90
opps! That was not a valid number. Try Again...

Please enter a number : 45

In [7]:
```

Exception Handling

ZeroDivisionError

exceptionDene2.py - F:/Lectures/Python/exceptionDene2.py

File Edit Format Run Options Windows Help

```
x, y = 5, 0

try:
    z = x / y
except ZeroDivisionError:
    print("Dikkat : 0 a bölünüyor")
```

Ln: 4 Col: 4

Exception Handling

ZeroDivisionError

exceptionDene1.py - F:\Lectures\Python\exceptionDene1.py

File Edit Format Run Options Windows Help

```
x, y = 5, 0
try:
    z = x//y
except ZeroDivisionError as e:
    print (e) # output: "integer division or modulo by zero"
```

Ln: 1 Col: 10

Exception Handling

IOError and ValueError

exceptionMyfile.py - F:/Lectures/Python/exceptionMyfile.py

File Edit Format Run Options Windows Help

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

Ln: 14 Col: 0

`try` Statement

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

exceptionFonksiyon.py - F:/Lectures/Python/exceptionFonksiyon.py

File Edit Format Run Options Windows Help

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Ln: 9 Col: 8

A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not.

`try` Statement

```
Python Shell
File  Edit  Shell  Debug  Options  Windows  Help
>>> ================================ RESTART ================================
>>>
>>> divide (2 ,1)
result is 2.0
executing finally clause
>>> divide (2 ,0)
division by zero!
executing finally clause
>>> divide (2 ,'0')
executing finally clause
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    divide (2 ,'0')
  File "F:/Lectures/Python/exceptionFonksiyon.py", line 3, in divide
    result = x / y
TypeError: unsupported operand type(s) for /: 'int' and 'str'
>>>
                                                              Ln: 122 Col: 4
```

As you can see, the `finally` clause is executed in any event. In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

try Statement

tryCatchDeneme2.py - D:/Lectures/BCO 601 Python Programming/tryCatchDeneme2.py (3.4.2)

File Edit Format Run Options Windows Help

```
import math

num = int(input('Enter number to compute factorial Of: '))
valid_input = False

while not valid_input:
    try:
        result = math.factorial(num)
        print(result)
        valid_input = True
    except ValueError:
        print('Cannot compute factorial of negative numbers')
        num = int(input('Please re-enter: '))
```

Ln: 14 Col: 0

Python 3.4.2 Shell

File Edit Shell Debug Options Windows Help

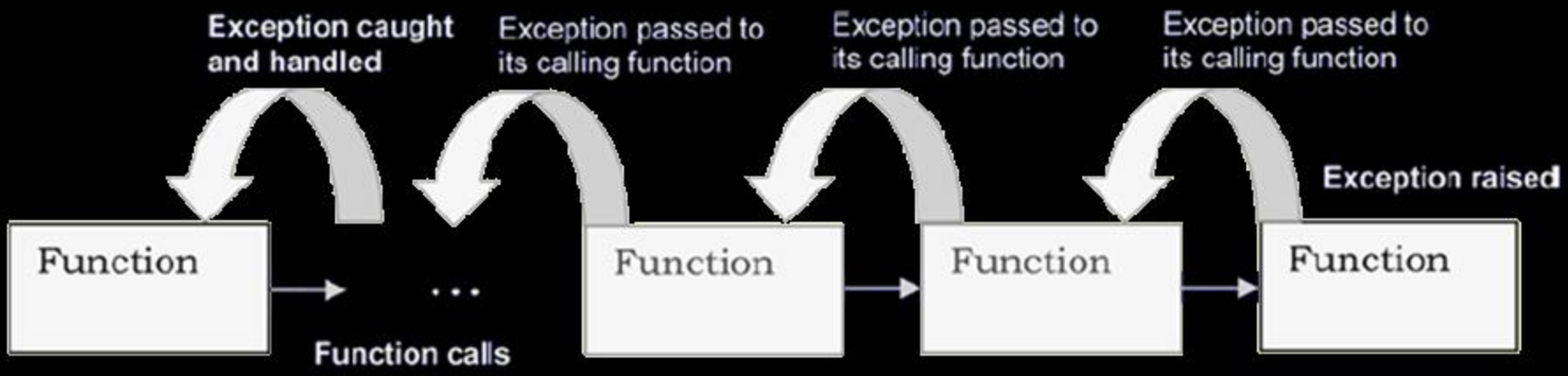
```
==============
>>>
Enter number to compute factorial Of: -5
Cannot compute factorial of negative numbers
Please re-enter: -3
Cannot compute factorial of negative numbers
Please re-enter: 3
6
>>>
```

Ln: 11 Col: 4

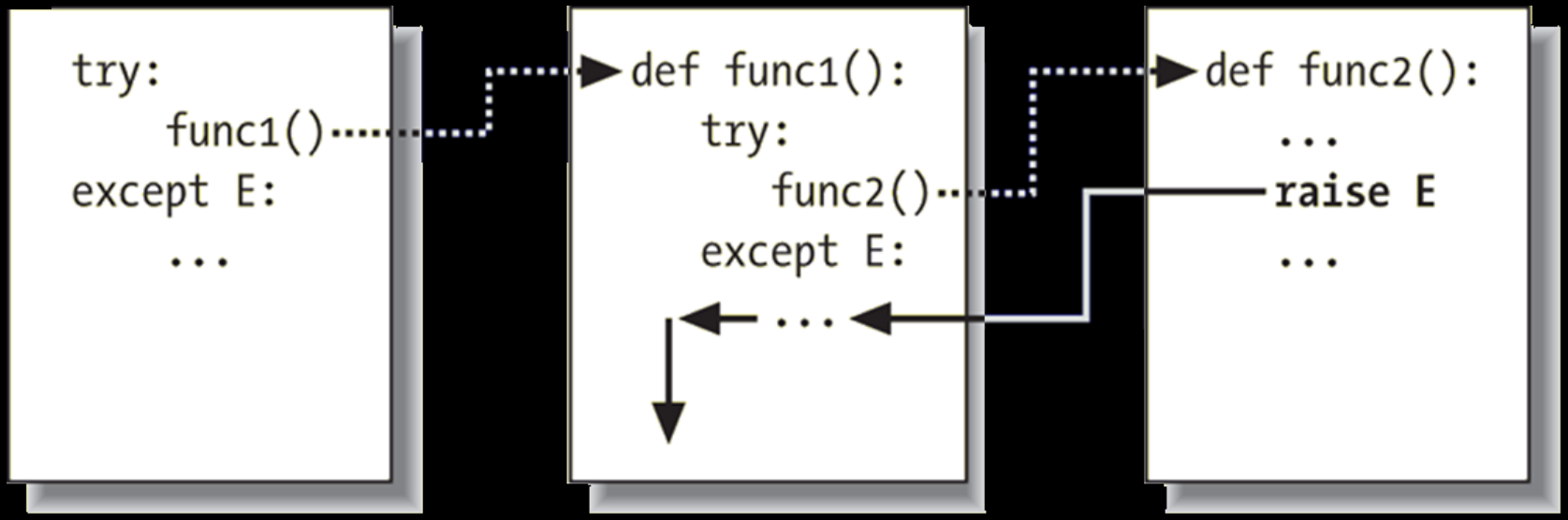
Nested try/catch Statement

```
def action2():
    print(1 + []) # Generate TypeError


def action1():
    try:
        action2()
    except TypeError: # Most recent matching try
        print('inner try')

try:
    action1()
except TypeError: # Here, only if action1 re-raises
    print('outer try')
```

Nested `try/catch` Statement

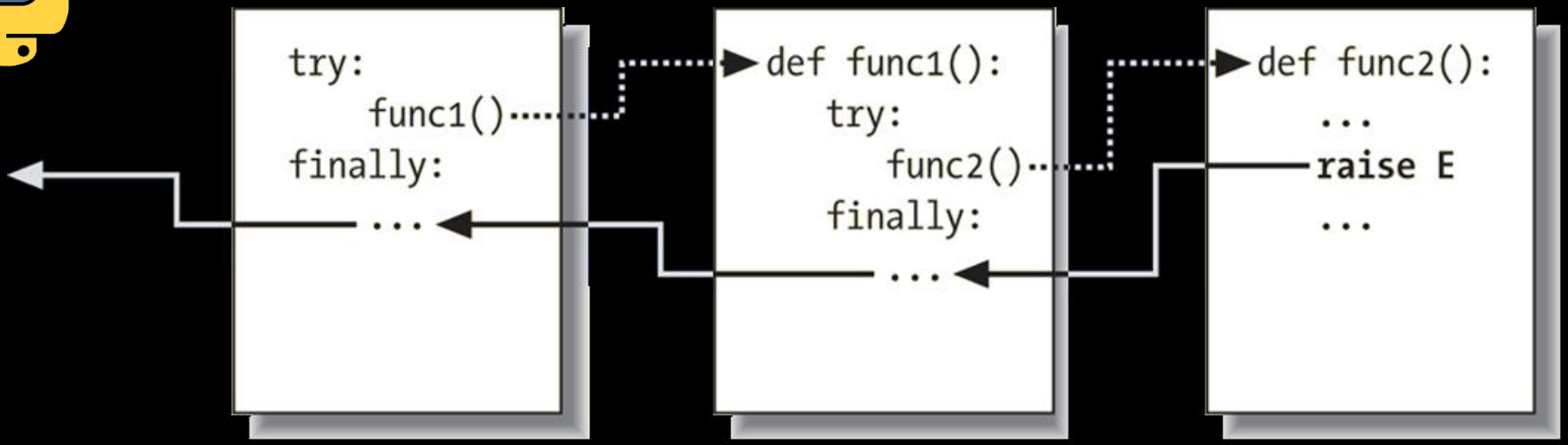


Nested `try/catch` Statement



when an exception is raised (by you or by Python), control jumps back to the most recently entered try statement with a matching except clause, and the program resumes after that try statement. except clauses intercept and stop the exception—they are where you process and recover from exceptions.

Nested `try/catch` Statement



when an exception is raised here, control returns to the most recently entered `try` to run its `finally` statement, but then the exception keeps propagating to all finallys in all active try statements and eventually reaches the default top-level handler, where an error message is printed. finally clauses intercept (but do not stop) an exception—they are for actions to be performed "on the way out."

Changing directory

Hint : Changing directory is comes in very handy when working in the Python interpreter:

```
>>> import os
>>> os.getcwd() # Returns the current working directory
'C:\\Python39'
# usually the directory you were in when you started the
interpreter

>>> os.chdir('/path/to/directory‘)
 # Change the current working directory to
'path/to/directory'
```

Working With Files

Python provides a built-in function open to open a file, which returns a file object

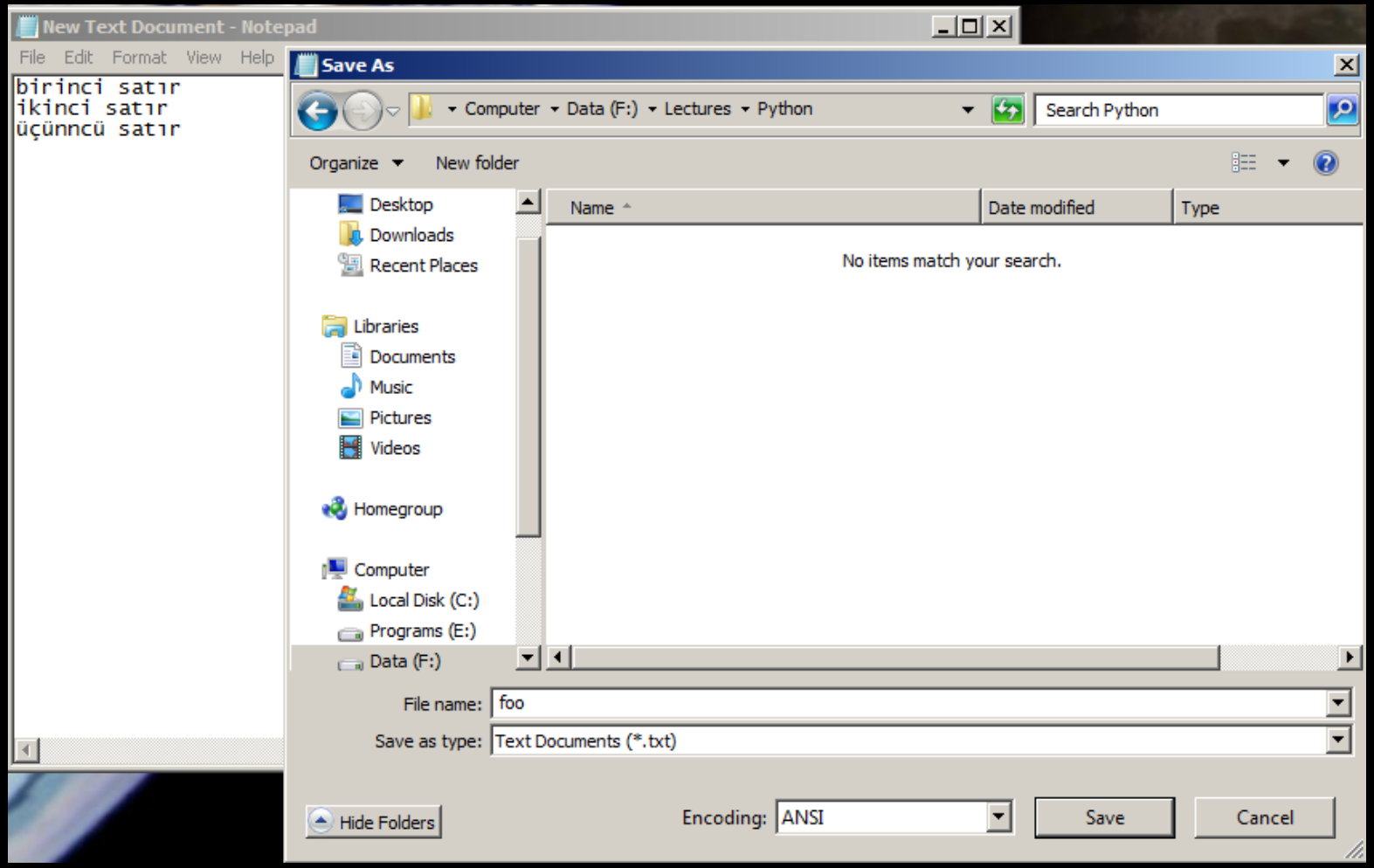
```
f = open('foo.txt', 'r') # open a file in read mode
f = open('foo.txt', 'w') # open a file in write mode
f = open('foo.txt', 'a') # open a file in append mode
```

'b' appended to the mode opens the file in binary mode: 'rb', 'wb', 'ab' should be used to open a binary file in read, write and append mode respectively. This mode should be used for all files that don't contain text.

`open(filename, mode)`

The mode argument is optional; 'r' will be assumed if it's omitted.

Working With Files



Working With Files

Easiest way to read contents of a file is by using the read method.

```
>>> open('foo.txt').read()
'birinci satir\nikinci satir\nüçüncü satir'
>>> f.close()

>>> print(open('foo.txt').read())
>>>
birinci satir
ikinci satir
üçüncü satir
>>>
```

Contents of a file can be read line-wise using `readline` and `readlines` methods.

```
>>> open('foo.txt').readlines()
['birinci satir\n', 'ikinci satir\n', 'üçüncü satir']
```

Working With Files

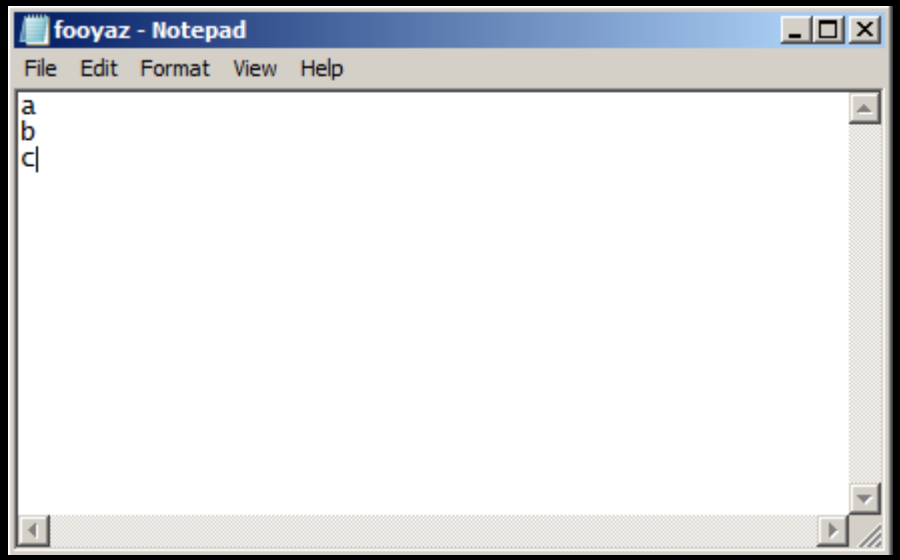
The `readline` method returns empty string when there is nothing more to read in a file.

```
>>> f = open('foo.txt')
>>> f.readline()
'birinci satir\n'
>>> f.readline()
'ikinci satir\n'
>>> f.readline()
'üçüncü satir\n'
>>> f.readline()
''
>>> f.close()
```

Working With Files

The write method is used to write data to a file opened in write or append mode

```
>>> f = open('fooyaz.txt','w')
>>> f.write('a\nb\nc')
>>> f.close()
```



Working With Files

The `writelines` method is convenient to use when the data is available as a list of lines.

```
>>> f = open('fooyaz.txt',`a')
>>> f.writelines(['a\n', 'b\n', 'c\n`])
>>> f.close()
```

Working With Files

Lets try to compute the number of characters, words and lines in a file.

Number of characters in a file is same as the length of its contents.

```
>>>def charcount(filename):
       return len(open(filename).read())
```

Number of words in a file can be found by splitting the contents of the file.

```
>>>def wordcount(filename):
       return len(open(filename).read().split())
```

Number of lines in a file can be found from `readlines` method

```
>>>def linecount(filename):
       return len(open(filename).readlines()
```

Working With Files

```
>>> import os
>>> os.getcwd()
'C:\\WPy64-3920\\Notebooks'
>>> os.chdir('C:\Lectures\HAB619')
>>> charcount('rapor.txt')
3206
>>> wordcount('rapor.txt')
205
>>> linecount('rapor.txt')
13
>>>
```