

# Modules – Exception Handling

## Performance Measuring

**#8**

Serdar ARITAN

Biomechanics Research Group,  
Faculty of Sports Sciences, and  
Department of Computer Graphics  
Hacettepe University, Ankara, Turkey



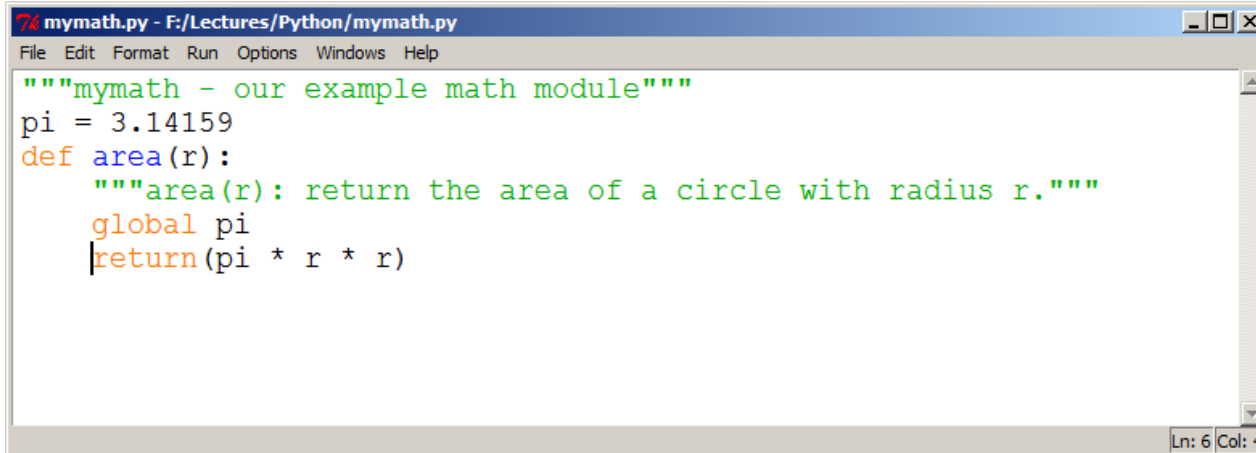
# Modules and Scoping Rules

Modules are used to organize larger Python projects. A module is a file containing code. A module defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

Modules most often contain Python source code, but they can also be compiled C or C++ object files. Compiled modules and Python source modules are used the same way.

For example, you might write a module for your program called **mymodule**, which defines a function called **reverse**. In the same program, you might also wish to use somebody else's module called **othermodule**, which also defines a function called **reverse**, but which does something different from your reverse function.

In a language without modules, it would be impossible to use two different functions named **reverse**. In Python, it's trivial—you refer to them in your main program as **mymodule.reverse** and **othermodule.reverse**. This is because Python uses namespaces. A namespace is essentially a dictionary of the identifiers available to a block, function, class, module, and so on. Write a module : **Create a text file called `mymath.py`, and in that text file enter the Python code in**



```
mymath.py - F:/Lectures/Python/mymath.py
File Edit Format Run Options Windows Help

"""mymath - our example math module"""
pi = 3.14159
def area(r):
    """area(r): return the area of a circle with radius r."""
    global pi
    return(pi * r * r)
```

Ln: 6 Col: 4



```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> pi
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>> area(2)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    area(2)
NameError: name 'area' is not defined
>>> |
```

In other words, Python doesn't have the constant `pi` or the function `area` built in.





```
Python Shell
File Edit Shell Debug Options Windows Help

>>>
>>> ===== RESTART =====
>>> pi
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>> import mymath
>>> pi
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    pi
NameError: name 'pi' is not defined
>>> mymath.pi
3.14159
>>> mymath.area(2)
12.56636
>>> mymath.__doc__
'mymath - our example math module'
>>> mymath.area.__doc__
'area(r): return the area of a circle with radius r.'
>>>
```

Ln: 65 Col: 4



```
Python Shell
File Edit Shell Debug Options Windows Help

>>> ===== RESTART =====
>>> from mymath import pi
>>> pi
3.14159
>>> area(2)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    area(2)
NameError: name 'area' is not defined
>>> mymath.area(2)
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    mymath.area(2)
NameError: name 'mymath' is not defined
>>> |
```

The name `pi` is now directly accessible because we specifically requested it using `from module import name`.

# Modules and Scoping Rules

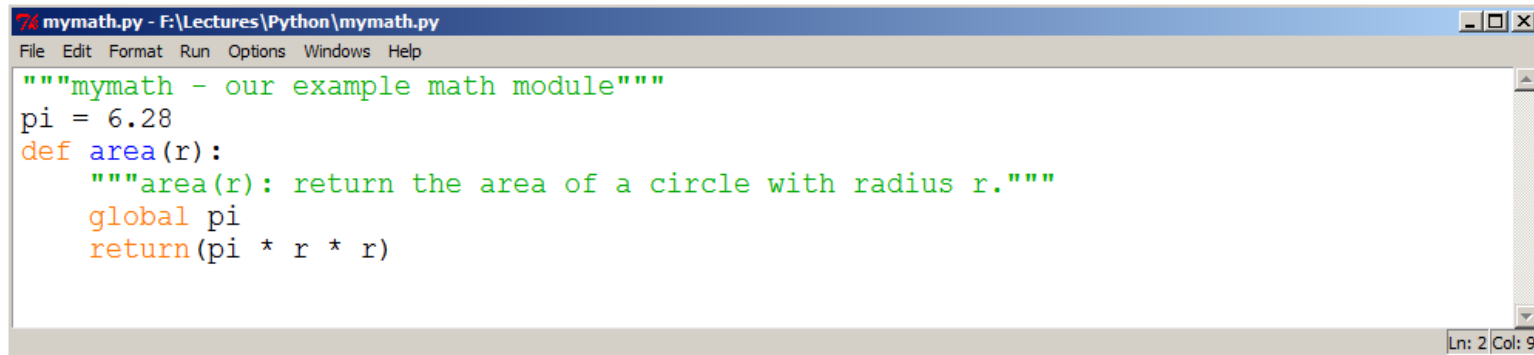
if you change your module on disk, retyping the **import** command won't cause it to load again. You need to use the **reload** function from the **imp** module for this.



```
Python Shell
File Edit Shell Debug Options Windows Help
>>> import mymath
>>> mymath.pi
3.14159
>>>
```

change the module on disk

import the **mymath** again



```
mymath.py - F:\Lectures\Python\mymath.py
File Edit Format Run Options Windows Help
"""mymath - our example math module"""
pi = 6.28
def area(r):
    """area(r): return the area of a circle with radius r."""
    global pi
    return(pi * r * r)
```



```
Python Shell
File Edit Shell Debug Options Windows Help
>>>
>>> import mymath
>>> mymath.pi
3.14159
>>> import mymath
>>> mymath.pi
3.14159
>>> import mymath, imp
>>> imp.reload(mymath)
<module 'mymath' from './mymath.py'>
>>> mymath.pi
6.28
>>>
```

When a module is reloaded (or imported for the first time), all of its code is parsed. A syntax exception is raised if an error is found. On the other hand, if everything is okay, a .pyc file (for example, mymath.pyc) containing Python byte code is created. Reloading a module doesn't put you back into exactly the same situation as when you start a new session and import it for the first time.



```
>>> import mymath, imp
```

```
Warning (from warnings module):
```

```
  File "<pyshell#11>", line 1
```



```
DeprecationWarning: the imp module is deprecated in  
favour of importlib and slated for removal in Python  
3.12; see the module's documentation for alternative  
uses
```

Python 3.4 and later which do have `importlib.reload`, use that instead.

# PYTHON PROGRAMMING

```
>>> import mymath, importlib
>>> importlib.reload(mymath)
>>> mymath.pi
>>> 3.141592
```



 mymath.cpython-39	10-Apr-21 1:07 PM	Compiled Pytho...	1 KB
 mymath.cpython-310	16-Nov-21 11:53 AM	Compiled Pytho...	1 KB

`.pyc` files are created by the Python interpreter when a `.py` file is imported. They contain the "compiled bytecode" of the imported module/program so that the "translation" from source code to bytecode (which only needs to be done once)





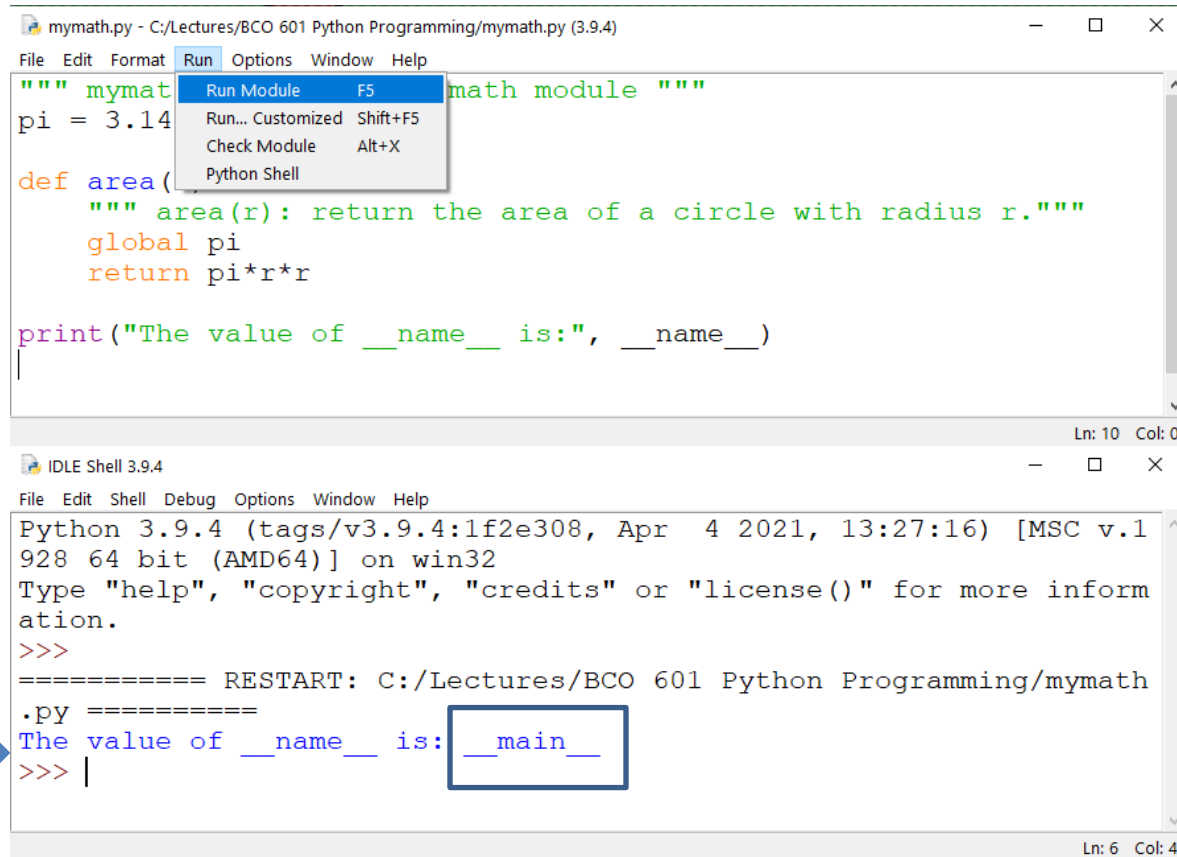
```
>>> import os
>>> os.getcwd()
'C:\\Python33\\Lib\\idlelib'
>>> os.chdir("F:\\Lectures\\Python")
>>> import mymath
>>> dir(mymath)
['__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 'area', 'pi']
>>> mymath.__file__
'\\.\\mymath.py'
>>> mymath.__name__
'mymath'
```

← `if __name__ == "__main__": ??`



# PYTHON PROGRAMMING

```
if __name__ == "__main__": main()
```



The screenshot shows the Python IDLE 3.9.4 environment. The top window, titled 'mymath.py - C:/Lectures/BCO 601 Python Programming/mymath.py (3.9.4)', contains the following code:

```
""" mymath module """
pi = 3.14

def area(r):
    """ area(r): return the area of a circle with radius r. """
    global pi
    return pi*r*r

print("The value of __name__ is:", __name__)
```

The bottom window, titled 'IDLE Shell 3.9.4', shows the output of running the script:

```
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr 4 2021, 13:27:16) [MSC v.1
928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inform
ation.
>>>
===== RESTART: C:/Lectures/BCO 601 Python Programming/mymath
.py =====
The value of __name__ is: __main__
>>> |
```

A blue arrow points from the text 'The value of \_\_name\_\_ is: \_\_main\_\_' in the shell output to the corresponding line in the script.



# PYTHON PROGRAMMING

```
if __name__ == "__main__": main()
```



```
IDLE Shell 3.9.4
File Edit Shell Debug Options Window Help
>>> import os
>>> os.getcwd()
'C:\Program Files\Python39\'
>>> os.chdir('C:\Lectures\BCO 601 Python Programming')
>>> os.getcwd()
'C:\Lectures\BCO 601 Python Programming'
>>> import mymath
The value of __name__ is: mymath
>>>
```

Ln: 8 Col: 0

```
Command Prompt - "c:\Program Files\Python39\python.exe"
C:\Lectures\BCO 601 Python Programming>"c:\Program Files\Python39\python.exe"
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr 4 2021, 13:27:16) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or ">>> license" for more information.
>>> import mymath
The value of __name__ is: mymath
>>>
```



```
mymath.py - C:\Lectures\BCO 601 Python Programming\mymath.py (3.9.4)
File Edit Format Run Options Window Help
""" mymath - our example math module """
pi = 3.141592

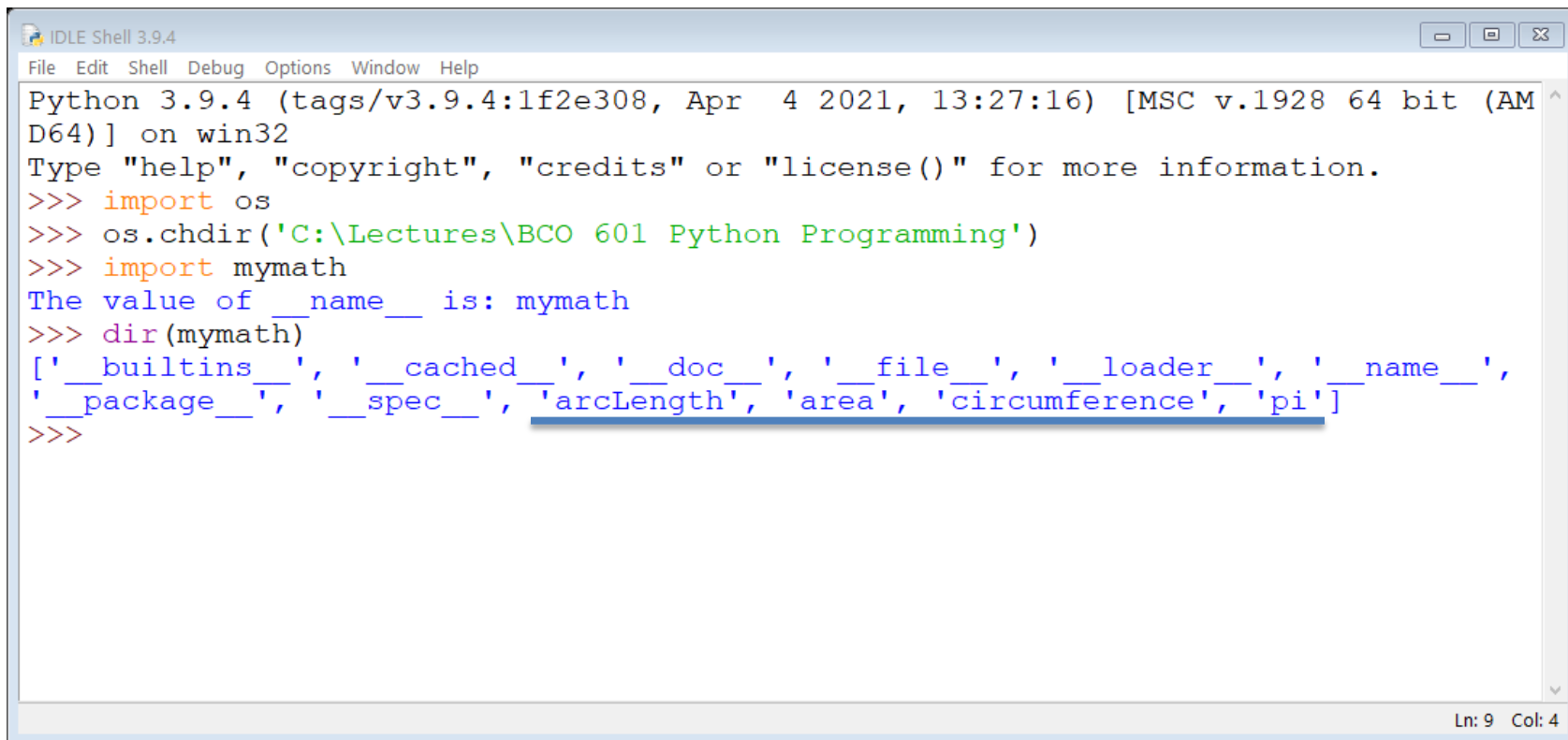
def area(r):
    """ area(r): return the area of a circle with radius r."""
    global pi
    return pi*r*r

def circumference(r):
    """ circumference(r): return the perimeter of a circle with radius r."""
    global pi
    return 2*pi*r

def arcLength(r, theta):
    """ arcLength(r, theta):return the arc length of a circle given radius and angle."""
    return r * 0.0174532925 * theta

print("The value of __name__ is:", __name__)

Ln: 15 Col: 88
```



```
IDLE Shell 3.9.4
File Edit Shell Debug Options Window Help
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  4 2021, 13:27:16) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.chdir('C:\Lectures\BCO 601 Python Programming')
>>> import mymath
The value of __name__ is: mymath
>>> dir(mymath)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'arcLength', 'area', 'circumference', 'pi']
>>>
```

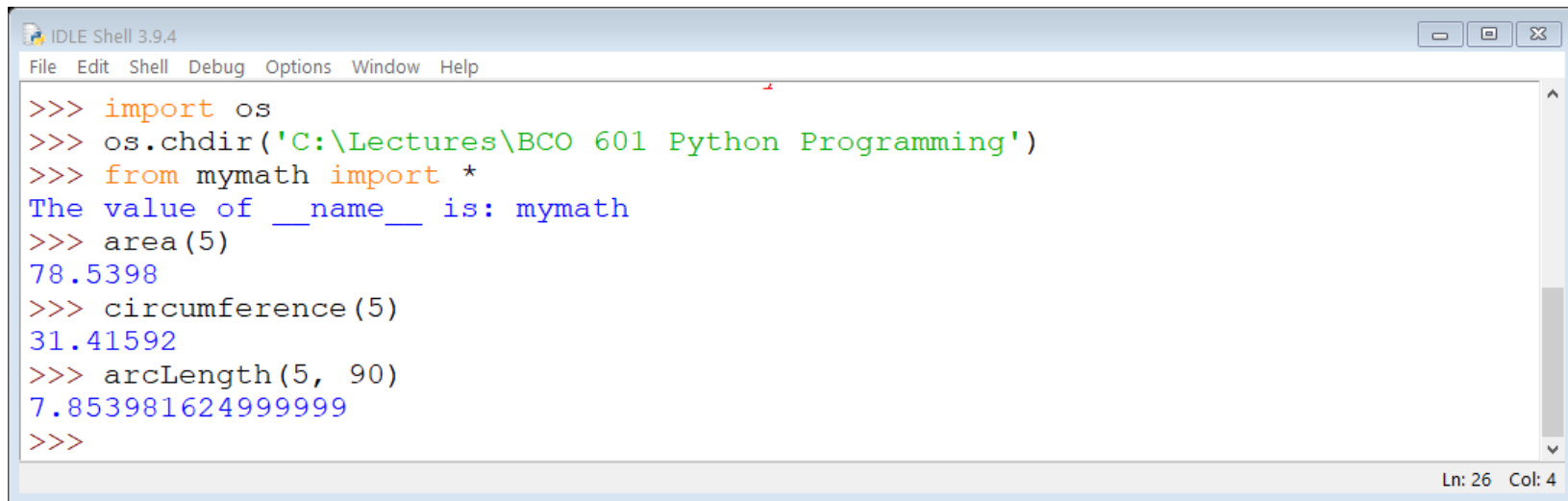
Ln: 9 Col: 4



```
*IDLE Shell 3.9.4*
File Edit Shell Debug Options Window Help
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  4 2021, 13:27:16) [MSC v.1928 64 bit (AMD64)] on win32
Type "help()" for more information.
>>> import os
>>> os.chdir('C:\\Users\\Serdar\\Documents\\BCO 601 Python Programming')
>>> import mymath
The value of the module is: mymath
>>> dir(mymath)
['__builtin__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'arcLength', 'area', 'circumference', 'pi']
>>> mymath.
```

Ln: 9 Col: 11





```
IDLE Shell 3.9.4
File Edit Shell Debug Options Window Help

>>> import os
>>> os.chdir('C:\Lectures\BCO 601 Python Programming')
>>> from mymath import *
The value of __name__ is: mymath
>>> area(5)
78.5398
>>> circumference(5)
31.41592
>>> arcLength(5, 90)
7.853981624999999
>>>
```

Ln: 26 Col: 4



```
mymath.py - C:\Lectures\BCO 601 Python Programming\mymath.py (3.9.4)
File Edit Format Run Options Window Help
""" mymath - our example math module """
pi = 3.141592

def area(r):
    """ area(r): return the area of a circle with radius r."""
    global pi
    return pi*r*r


def circumference(r):
    """ circumference(r): return the perimeter of a circle with radius r."""
    global pi
    return 2*pi*r

def arcLength(r, theta):
    """ arcLength(r, theta): return the arc length of a circle given radius and a
    return r * 0.0174532925 * theta

__all__ = ['area', 'circumference']
if __all__ is defined, then only the names
explicitly listed will be exported
print("The value of __name__ is:", __name__)
```

Not Listed

Ln: 18 Col: 2



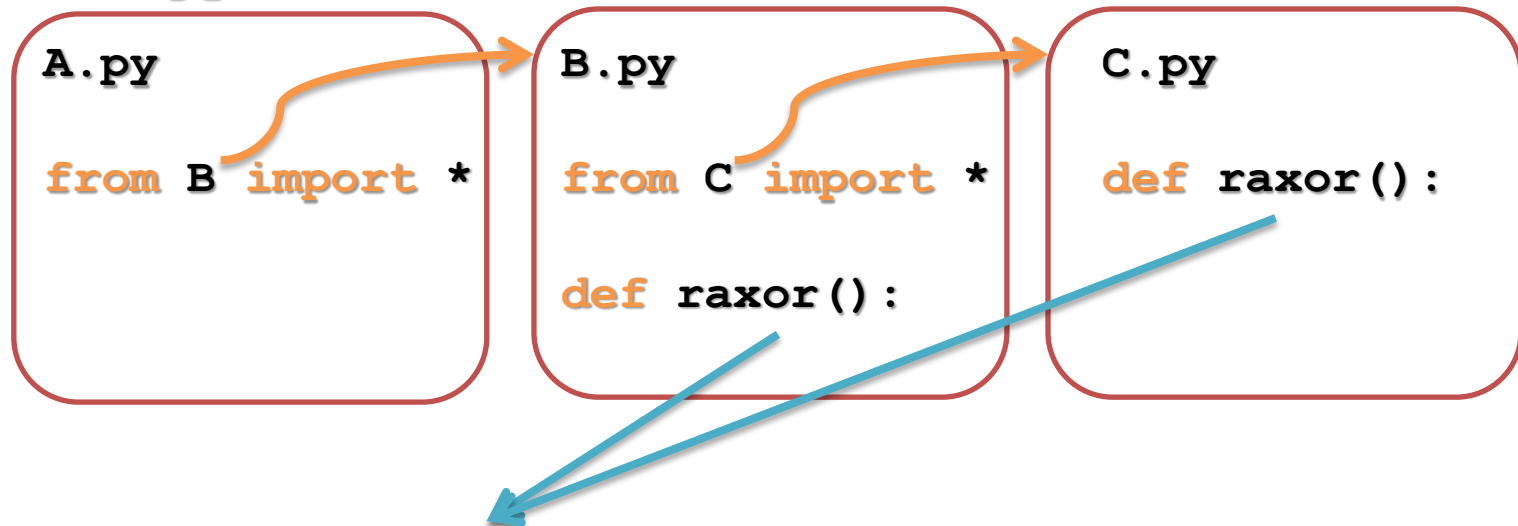
```
IDLE Shell 3.9.4
File Edit Shell Debug Options Window Help
>>> import os
>>> os.chdir('C:\Lectures\BCO 601 Python Programming')
>>> from mymath import *
The value of __name__ is: mymath
>>> area(5)
78.5398
>>> circumference(5)
31.41592
>>> arcLength(5, 90)
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    arcLength(5, 90)
NameError: name 'arcLength' is not defined
>>>
```

`__all__ = ['area', 'circumference']`

Ln: 61 Col: 4

# Modules and Scoping Rules

Import can be used within modules as well as in the main program. The main program could be `A.py`, which has an import statement “`from B import *`” implying there is a file `B.py`. The file `B.py` could also have an import statement, such as “`from C import *`” to bring definitions in from `C.py`.



**Same name but does something completely different!!**



## Modules and Scoping Rules

The general answer is that the second module to be imported overwrites the definitions of the first module that was imported. An import statement “**from C import \***” overwrites any defined functions or variables that have the same names as the ones in **C.py**.

**Most often, the people who write programs importing modules do not know all the function names inside a module**, and really are only interested in a few things. Suppose that a main program only needs a function **capit** from **C.py**. even if **C.py** has a **raxor** function that would overwrite the one **B.py** has defined, there is no conflict: only the function **capit** is brought in from **C.py**.

```
from B import *  
from C import capit
```

module1.py

```
def double(lst):  
    '''returns a new list  
    each number doubled  
    [1 2 3] returned as  
    [2 4 6]'''
```

module2.py

```
def double(lst):  
    '''returns a new list  
    with each number  
    duplicated [1 2 3]  
    returned as  
    [(1,1) (2, 2) (3, 3)]'''
```

main.py

```
from module1 import *  
from module2 import *  
num_list = [3, 8, 14]  
result = double(num_list)
```





With this style of importing, a script could use two functions named `foo()` found in different modules. For example, there could be modules `A`, `B`, `C` imported in a script. The print can refer to the three different `foo` functions without any name conflict.

```
import A
import B
import C
```

```
print(A.foo(), B.foo(), C.foo())
```

Enter each of the following functions in their own modules named `mod1.py` and `mod2.py`. Enter and execute the following and observe the results.

```
# mod1
def average(lst):
    print('average of mod1 called')

# mod2
def average(lst):
    print('average of mod2 called')

>>> import mod1, mod2
>>> mod1.average([10, 20, 30])
???
>>> mod2.average([10, 20, 30])
???
>>> average([10, 20, 30])
???
```

Many Python modules and main programs use **import** statements for more than one module. A typical example, shown here, imports typical modules from the standard library. The example shows that four modules are imported, though perhaps more are imported indirectly, behind the scene, because modules may import modules. The same example could be more concisely expressed as

```
import sys      import sys, os      import sys, os, csv, time
import os       import csv, time
import csv
import time
```

All of these forms of using **import** get the same result.



Exactly where Python looks for modules is defined in a variable called `path`, which you can access through a module called `sys`. Enter the following:

```
>>> import sys
>>> sys.path
['C:/Lectures/BCO 601 Python Programming/Python3_13',
'C:\\Program Files\\Python313\\Lib\\idlelib', 'C:\\Program
Files\\Python313\\python313.zip', 'C:\\Program
Files\\Python313\\DLLs', 'C:\\Program Files\\Python313\\Lib',
'C:\\Program Files\\Python313', 'C:\\Program
Files\\Python313\\Lib\\site-packages']
```

Python searches (in order) when attempting to execute an import statement. The first module found that satisfies the import request is used. If there's no satisfactory module in the module search path, an `ImportError` exception is raised.



`append()` is a built-in function of `sys` module that can be used with path variable to add a specific path for interpreter to search:

```
>>> sys.path.append("C:\\Users\\SA-Lenovo\\Desktop")
>>> sys.path
['C:/Lectures/BCO 601 Python Programming/Python3_13',
 'C:\\Program Files\\Python313\\Lib\\idlelib', 'C:\\Program
Files\\Python313\\python313.zip', 'C:\\Program
Files\\Python313\\DLLs', 'C:\\Program Files\\Python313\\Lib',
 'C:\\Program Files\\Python313', 'C:\\Program
Files\\Python313\\Lib\\site-packages',
 'C:\\Users\\SA-Lenovo\\Desktop']
```





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.



### 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 parameters.





### 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`.

```
def 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...
```





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.



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.



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:

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

### Different exception types

```
Python Shell
File Edit Shell Debug Options Windows Help

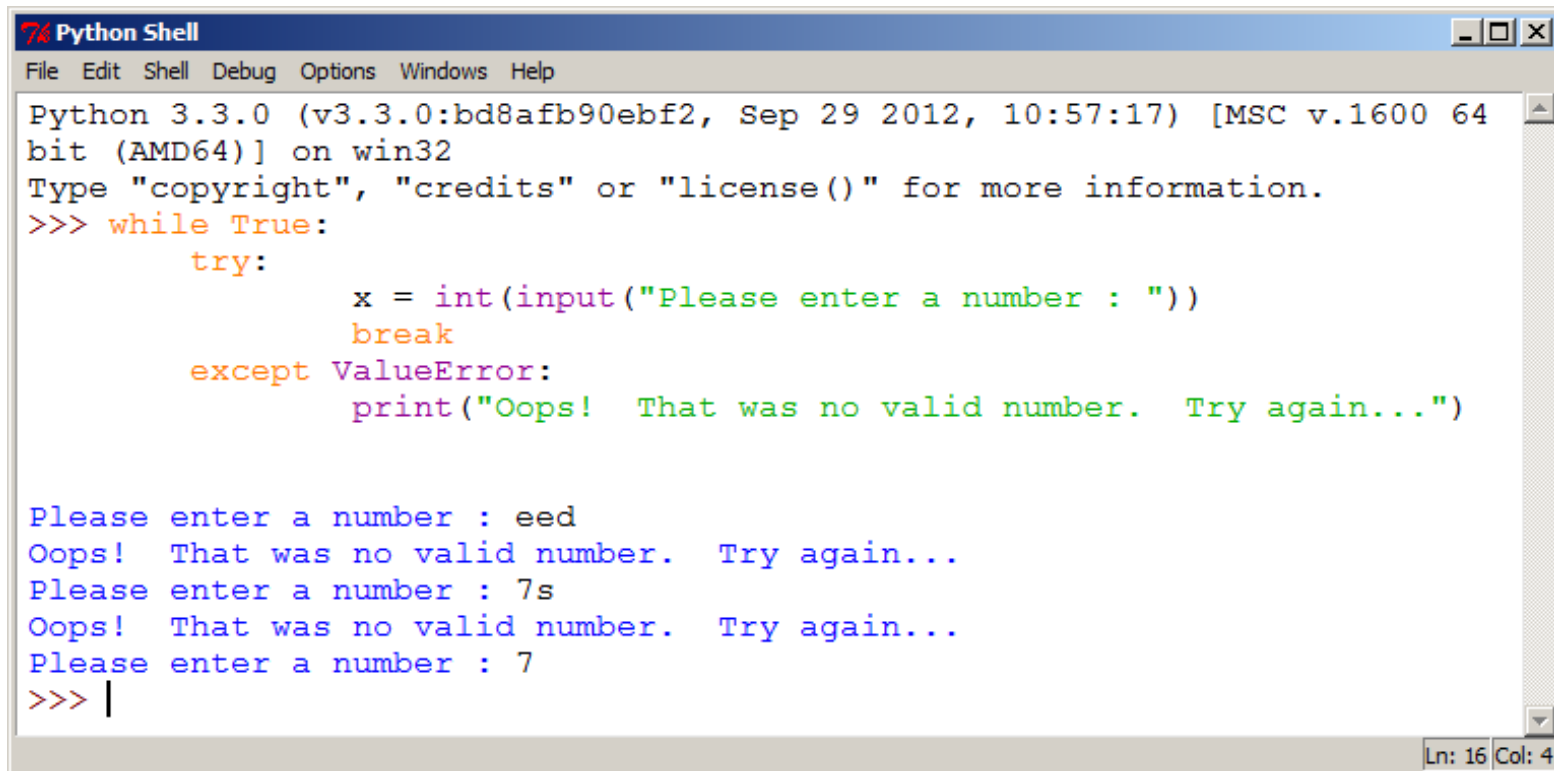
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    10 * (1/0)
ZeroDivisionError: division by zero

>>> 4+spam*3
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    4+spam*3
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    '2' + 2
TypeError: Can't convert 'int' object to str implicitly

Ln: 45 Col: 0
```

### ValueError : input only number example

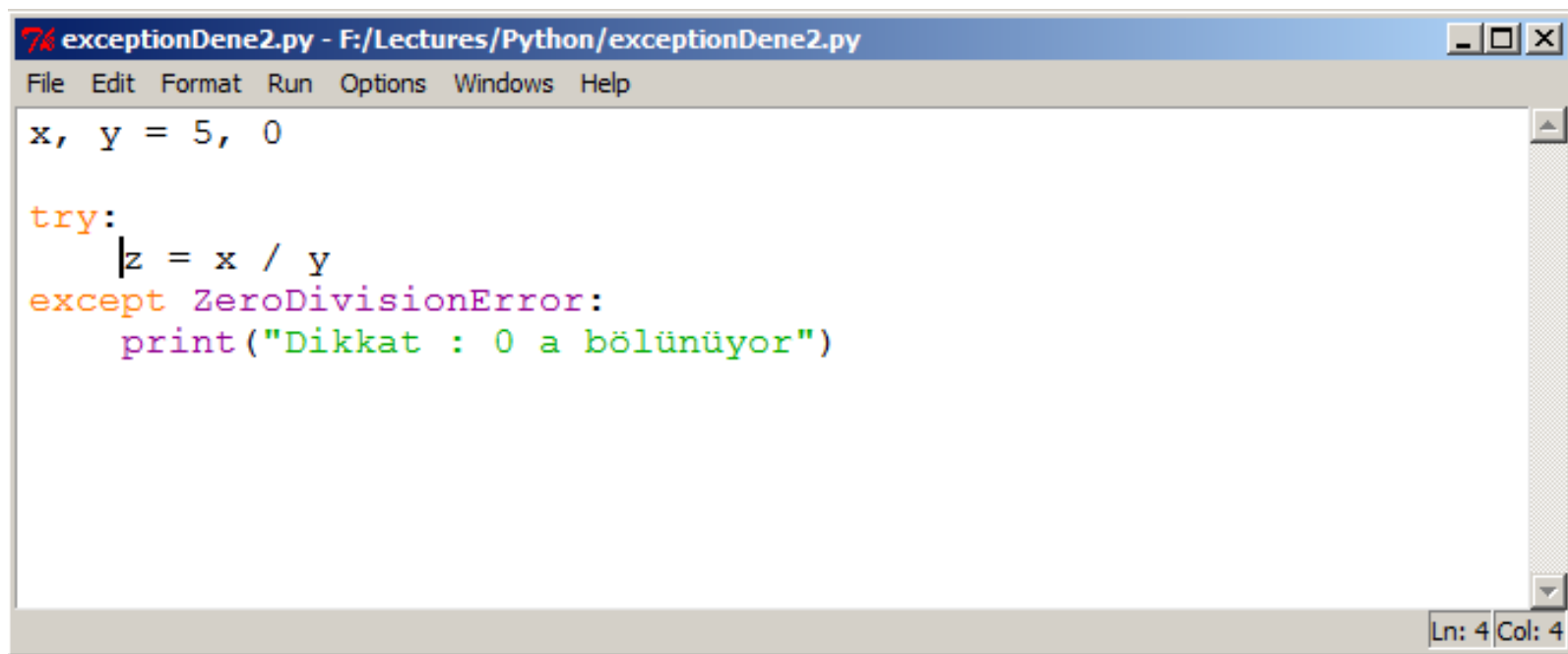


```
Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64
bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> while True:
    try:
        x = int(input("Please enter a number : "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")

Please enter a number : eed
Oops! That was no valid number. Try again...
Please enter a number : 7s
Oops! That was no valid number. Try again...
Please enter a number : 7
>>> |
```

Ln: 16 Col: 4

### ZeroDivisionError :



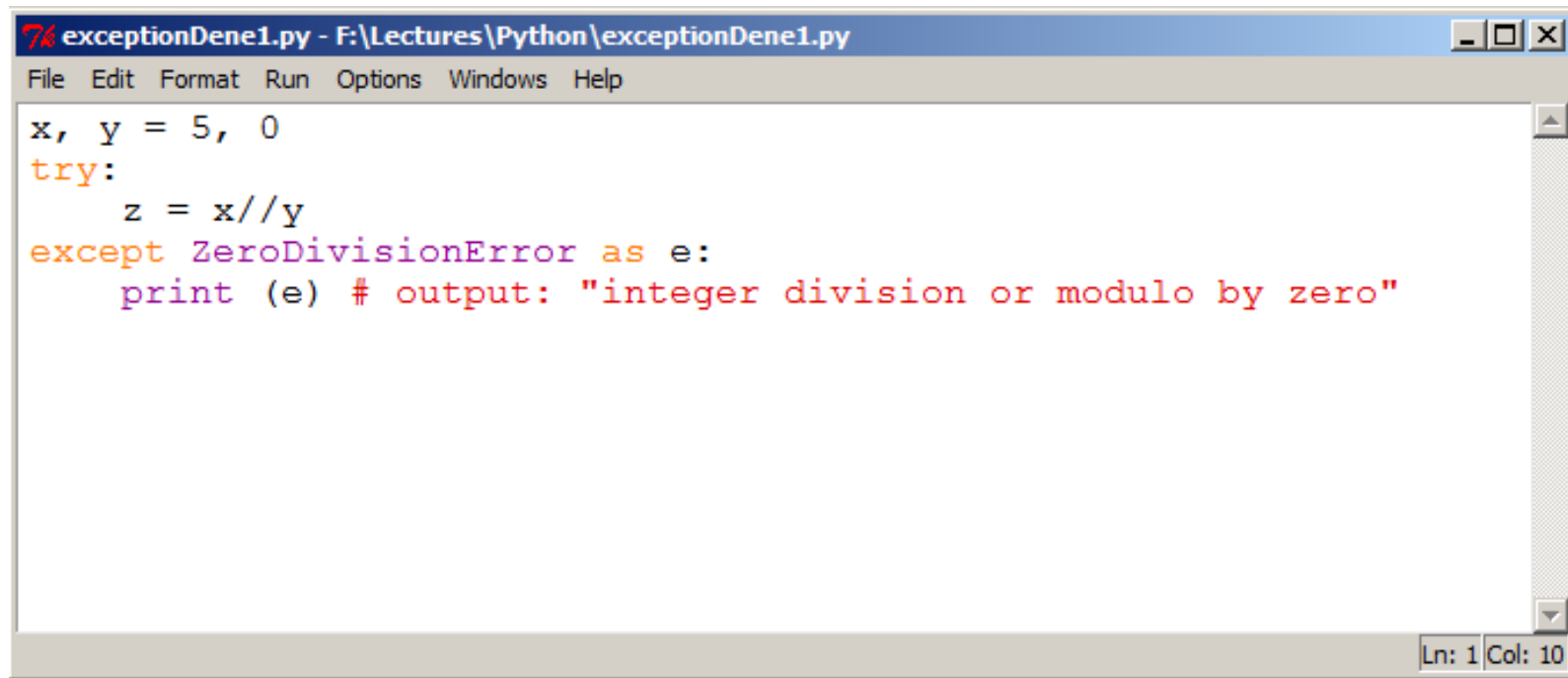
```
74 exceptionDene2.py - F:/Lectures/Python/exceptionDene2.py
File Edit Format Run Options Windows Help

x, y = 5, 0

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

Ln: 4 Col: 4
```

### 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
```

## ZeroDivisionError :

Let's create a simple calculator that does only

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    answer = int(first_number) / int(second_number)
    print(answer)
```



## ZeroDivisionError :

```
Give me two numbers, and I'll divide them.  
Enter 'q' to quit.
```

```
First number: 5
```

```
Second number: 0
```

```
Traceback (most recent call last):
```

```
  File "C:/Lectures/BCO 601 Python Programming/myDivision.py",  
    line 10, in <module>
```

```
    answer = int(first_number) / int(second_number)
```

```
ZeroDivisionError: division by zero
```

## ZeroDivisionError :

```
print("Give me two numbers, and I'll divide them.")
print("Enter 'q' to quit.")
while True:
    first_number = input("\nFirst number: ")
    if first_number == 'q':
        break
    second_number = input("Second number: ")
    if second_number == 'q':
        break
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("You can't divide by 0!")
    else:
        print(answer)
```

### ZeroDivisionError :

```
Give me two numbers, and I'll divide them.  
Enter 'q' to quit.
```

```
First number: 5  
Second number: 0  
You can't divide by 0!
```

```
First number: 5  
Second number: 2  
2.5
```

```
First number: q
```

Handling the **FileNotFoundError** Exception

One common issue when working with files is handling missing files. The file you're looking for might be in a different location, the filename may be misspelled, or the file may not exist at all.

```
filename = 'alice.txt'
```

```
with open(filename, encoding='utf-8') as f:  
    contents = f.read()
```

```
Traceback (most recent call last):
```

```
  File "C:/Lectures/BCO 601 Python Programming/alice.py", line 3,  
in <module>
```

```
    with open(filename, encoding='utf-8') as f:  
FileNotFoundError: [Errno 2] No such file or directory:  
'alice.txt'
```



Handling the **FileNotFoundError** Exception

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f'Sorry, the file {filename} does not exist.')
```

Sorry, the file alice.txt does not exist.

Handling the **FileNotFoundError** Exception

```
filename = 'alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f'Sorry, the file {filename} does not exist.')
else:
    # Count the approximate number of words in the file.
    words = contents.split()
    num_words = len(words)
    print(f'The file {filename} has about {num_words} words.')
```

The file `alice.txt` has about 1616 words.

Handling the **FileNotFoundError** Exception

```
def count_words(filename):  
    """Count the approximate number of words in a file."""  
    try:  
        with open(filename, encoding='utf-8') as f:  
            contents = f.read()  
    except FileNotFoundError:  
        print(f'Sorry, the file {filename} does not exist.')  
    else:  
        # Count the approximate number of words in the file.  
        num_words = len(contents.split())  
        print(f'The file {filename} has about {num_words} words.')
```

Handling the **FileNotFoundError** Exception

```
filenames=['alice.txt','siddhartha.txt','moby_dick.txt', 'little_women.txt']  
for filename in filenames:  
    count_words(filename)
```

The file `alice.txt` has about 1616 words.

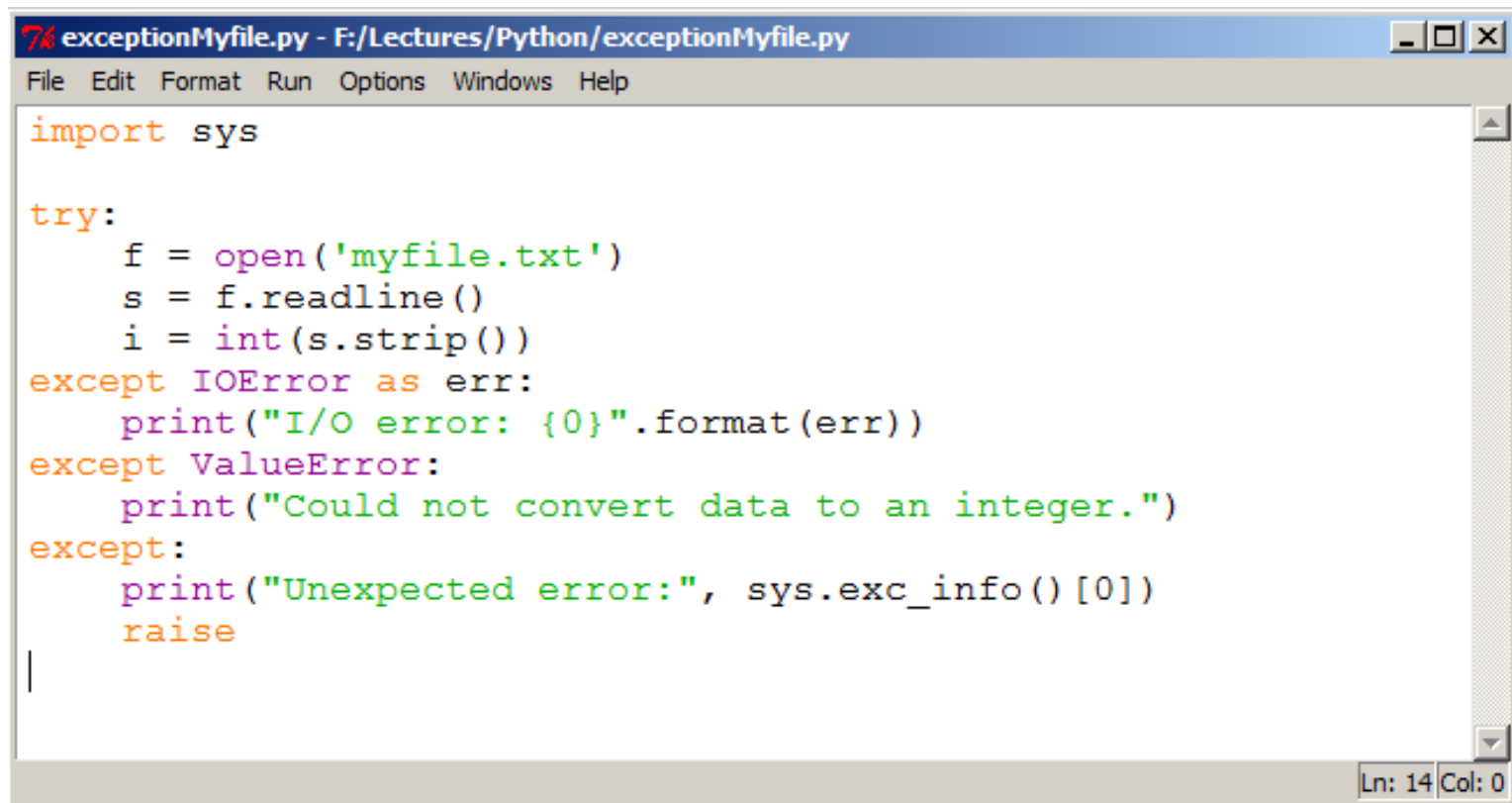
Sorry, the file `siddhartha.txt` does not exist.

The file `moby_dick.txt` has about 215830 words.

The file `little_women.txt` has about 189079 words.



### 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: {}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise

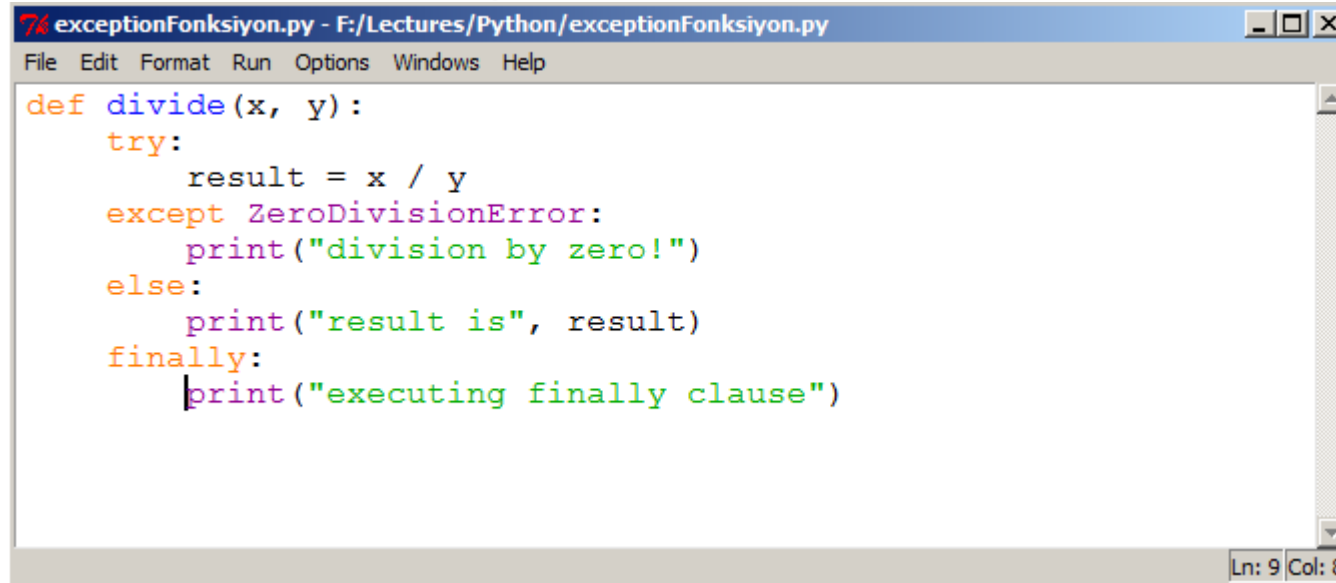
Ln: 14 Col: 0
```



# PYTHON PROGRAMMING

## 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:



```
7% 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
```

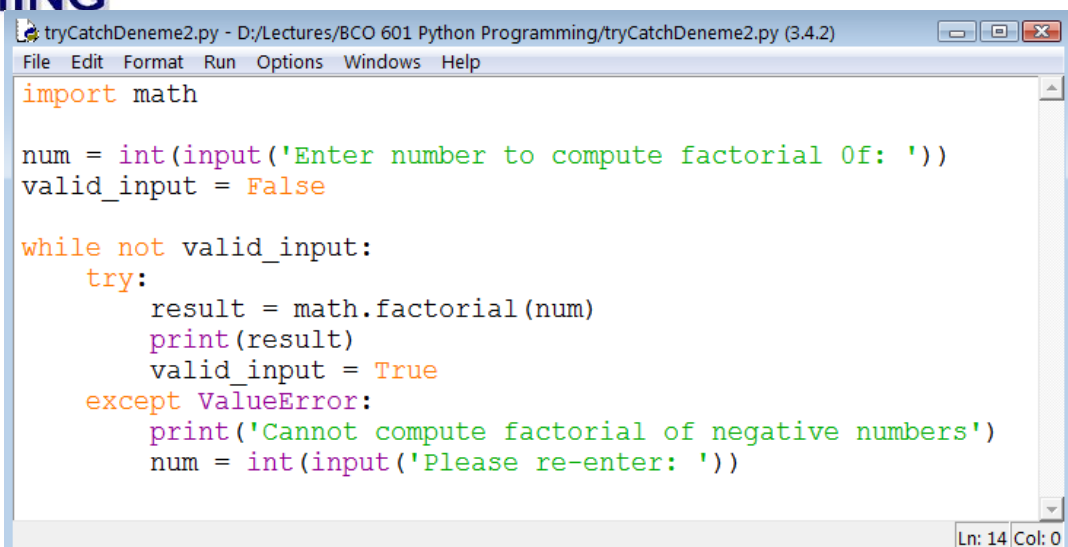
A `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not.



```
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'
>>> |
```

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.



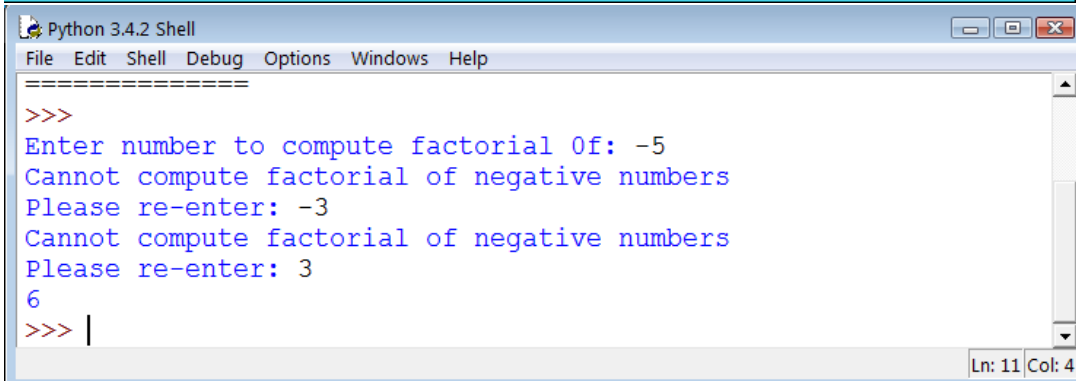
```
tryCatchDeneme2.py - D:/Lectures/BCO 601 Python Programming/tryCatchDeneme2.py (3.4.2)
File Edit Format Run Options Windows Help

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

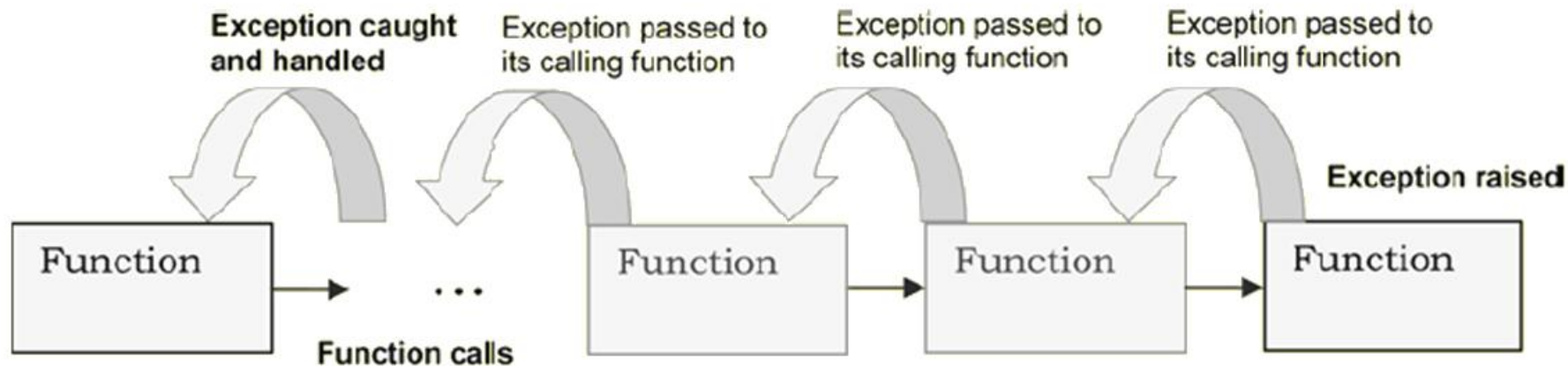
=====
>>>
Enter number to compute factorial Of: -5
Cannot compute factorial of negative numbers
Please re-enter: -3
Cannot compute factorial of negative numbers
Please re-enter: 3
6
>>> |

Ln: 11 Col: 4
```



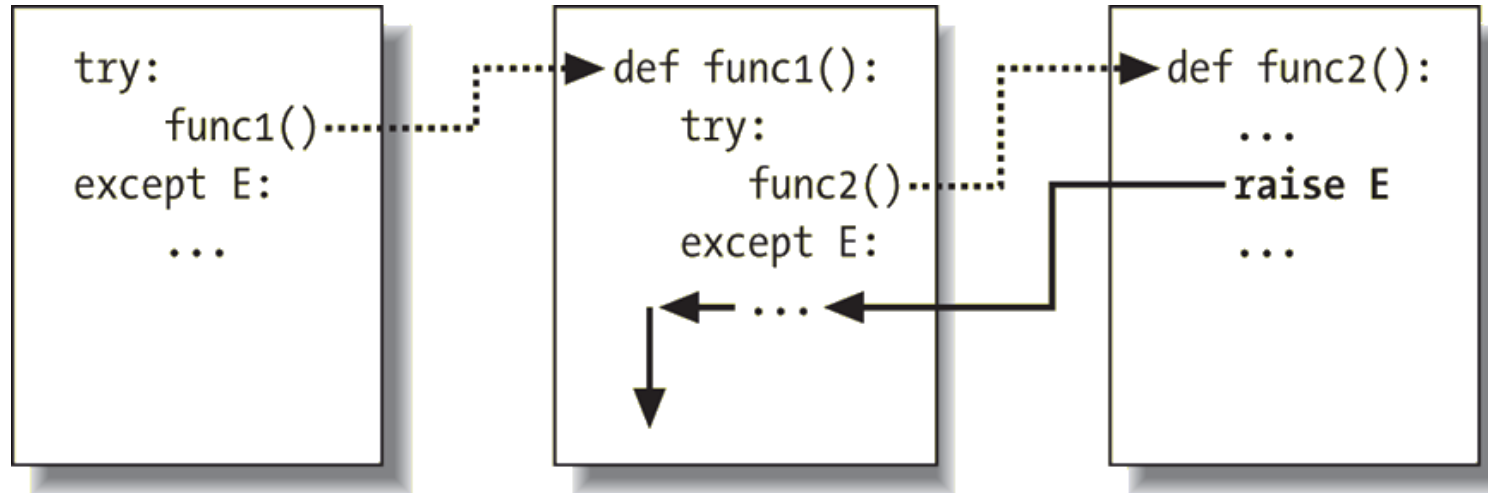
```
def action2():  
    print(1 + []) # Generate TypeError  
  
def action1():  
    try:  
        action2()  
    except TypeError: # Most recent matching try  
        print('inner try')  
  
try:  
    action1()  
except TypeError: # Here, only if action1 re-raises  
    print('outer try')
```

# Nested try/catch Statement





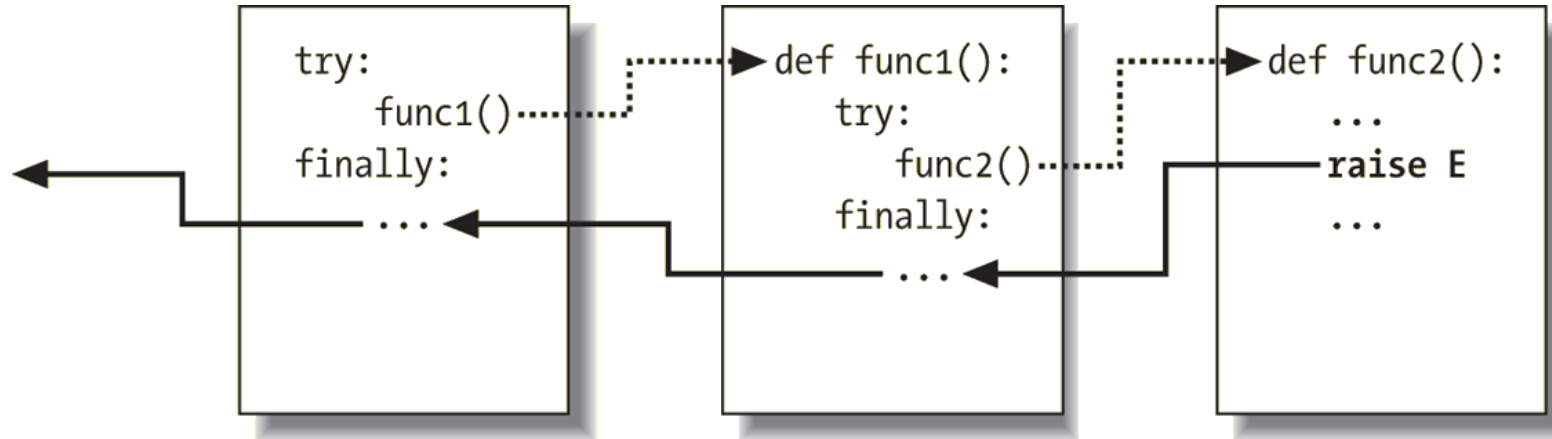
## Nested try/catch Statement



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 `finally`s 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.”





`time` module provides various time-related functions.

```
>>> from time import gmtime, strftime, perf_counter()
```

```
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())  
'Sat, 26 Mar 2022 10:59:51 +0000'
```

```
>>> perf_counter()
```

```
32311.48899951
```

```
>>> perf_counter()  # A few seconds later
```

```
32315.261320793
```

```
import time

n = int(1e6)
tic = time.perf_counter()
top = 0
for i in range(n):
    top = top + i**3
print(top)
toc = time.perf_counter()
print(f"The sum of power of 3 the numbers between 0 to {n} took {toc - tic:0.4f} seconds")

tic = time.perf_counter()
print(sum([x**3 for x in range(n)]))
toc = time.perf_counter()
print(f"The sum of power of 3 the numbers between 0 to {n} took {toc - tic:0.4f} seconds")

2499995000002500000000000
The sum of power of 3 the numbers between 0 to 1000000 took 0.3712 seconds
2499995000002500000000000
The sum of power of 3 the numbers between 0 to 1000000 took 0.2862 seconds
```



```
>>> numbers = "-".join(str(n) for n in range(100))
>>> numbers
'0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26-27-28-29-30-
31-32-33-34-35-36-37-38-39-40-41-42-43-44-45-46-47-48-49-50-51-52-53-54-55-56-57-58-
59-60-61-62-63-64-65-66-67-68-69-70-71-72-73-74-75-76-77-78-79-80-81-82-83-84-85-86-
87-88-89-90-91-92-93-94-95-96-97-98-99'

>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

```
import timeit
```

```
#code snippet
```

```
code = "[i for i in range(199)]"
```

```
#execute the code 1 time
```

```
execution_time = timeit.timeit(stmt = code, number =1)
```

```
print(execution_time, "seconds" )
```

---

```
import timeit
```

```
#code snippet
```

```
code = "a= 20;b= 35;c=39;sum= a+b+c"
```

```
#execute the code 1000 time
```

```
execution_time = timeit.timeit(stmt = code)
```

```
print("Total Execution time:",execution_time, "seconds" )
```

```
#divide the execution_time with number or 1000000 to findout the best execution time  
of code
```

```
print("Single Execution time:", execution_time/1000000, "seconds")
```

```
import timeit

#code snippet
code = """
def sum(a,b,c):
    return a+b+c
"""

#execute the code 100000 time
execution_time = timeit.timeit(stmt = code, number =100000)
print("Total Execution time:",execution_time, "seconds" )

#divide the execution_time with number or 100000 to findout the best execution time of
code
print("Single Execution time:", execution_time/100000, "seconds")
```

```
import timeit

#setup code
import_math = "import math"

#code snippet
code = """
def sum(a,b,c):
    return (math.sqrt(a) + math.sqrt(b) + math.sqrt(c))**2
"""

#execute the code 200000 time
execution_time = timeit.timeit(stmt = code, setup= import_math, number =200000)
print("Total Execution time:",execution_time, "seconds" )

#divide the execution_time with number or 200000 to findout the best execution time of
code
print("Single Execution time:", execution_time/200000, "seconds")
```