A SunCam online continuing education course

Python Programming for Engineers - Part 2: Branching and Looping, Functions and Error Handling

by

Kwabena Ofosu, Ph.D., P.E., PTOE

Computer Programming in *Python* – Part 2
A SunCam online continuing education course

Abstract

Python is a widely used, free, open source, high-level, general purpose computer programming language. *Python* drives some of the internet's most popular websites such as *Google*, *Youtube* and *Instagram*. *Python* can be used to perform complex mathematical calculations, handle big data, build web apps and desktop applications, and manage databases.

This course is the second of a series on *Python* programming. This course is tailored to practicing engineers. In this course, the following topics are presented in detail: conditional statements, looping structures, functions, modules, input and out (I/O) functions, file handling, and error handling techniques. Practical examples from situations encountered by practicing engineers and scientists are used to illustrate and demonstrate the concepts and methods learned in this course.

On completion of this course, participants will be capable of applying the methods and techniques learned in a desktop application that can be used to manage large data sets and automate complex, repetitive, and tedious engineering calculations and algorithms. Participants will be able to identify professional situations in which programming will be of a strategic advantage to them in their fields of specialty, and to their organizations. Programming continues to be an increasingly relevant and advantageous skill for engineers competing in a global marketplace in the computer age.

There are no required pre-requisites for this course. However, it will be helpful to understand the fundamentals of the *Python* programming language in general, as presented in the earlier parts of this course series.

TABLE OF CONTENTS

Abstract ii
List of Tables vi
1. INTRODUCTION 1
  1.1 Computer Programming 1
  1.2 Relevance of Computer Programming to Engineers 1
  1.3 Python 1
2. CONDITIONAL STATEMENTS 3
  2.1 Definition 3
  2.2 The *if* Statement 3
  2.3 The *if…else* Statement 12
  2.4 The *if…elif…else* Statement 12
  2.5 Short Hand Syntax 13
  2.6 The *pass* Statement 14
  2.7 Composite Conditional Expressions 14
  2.8 Nested Conditional Statements 16
3. LOOPING 19
  3.1 Definition 19
  3.2 The *while* Loop 19
  3.3 Nested Loops 24
  3.4 Infinite Loops 26
  3.5 The *break* Statement 27
  3.6 The *continue* Statement 28
  3.7 The *else* Statement 28
  3.8 Short Hand Syntax 29
  3.9 The *for* Loop 29
  3.10 The *range( )* Function 30

4. FUNCTIONS 35
4.1 Definition 35
4.2 Creating a Function 35
4.3 Calling a Function 35
4.4 Arguments 36
4.5 Default Arguments 36
4.6 Keyword Arguments 37
4.7 Returning a Value 39
4.8 Scope of a Variable 40
4.9 Recursion 53
4.10 Lambda Functions 54
5. INPUT AND OUTPUT (I/O) FUNCTIONS 56
5.1 Introduction 56
5.2 The *input( )* Function 56
5.3 Casting 57
5.4 The *print( )* Function 57
6. MODULES 66
6.1 Introduction 66
6.2 Calling a Module 66
6.3 Naming and Renaming a Module 66
6.4 Variables in Modules 67
6.5 Built-in in Modules 67
6.6 The *dir( )* Function 68
6.7 The *from* Keyword 68
7. FILE HANDLING 72
7.1 Introduction 72
7.2 The *open( )* Function 72
7.3 The *read( )* Method 73
7.4 The *readline( )* Method 74
7.5 The *close( )* Method 74
7.6 The *write( )* Method 75

7.7 Deleting Files 75
7.8 Renaming Files 76
7.9 Deleting a Folder 76
8. ERROR HANDLING AND EXCEPTIONS 87
8.1 Errors 87
8.2 Types of Errors 87
8.3 Handling Exceptions in *Python* 88
8.4 The *try* … *except* Clause 90
8.5 The *try* … *except* Clause for Multiple Exceptions 90
8.6 The *else* Keyword 91
8.7 The *finally* Block 91
8.8 The *raise* Keyword 92
8.9 Debugging *Python* Scripts 104
8.10 Getting Help 104
9. CONCLUSION 105
REFERENCES 106

List of Tables

Table 2. 1: Logical operators 4

Computer Programming in *Python* – Part 2
A SunCam online continuing education course

1. INTRODUCTION

1.1 Computer Programming

A **programming language** is a formal language specifically designed to communicate instructions to a computer. Programming languages can be used for many purposes, for example, to create programs that control the behavior of a computer or device, and/or to implement algorithms accurately and efficiently.

Computer programming (or programming) is a comprehensive process of formulating a computing problem, developing a methodology to solve the problem, writing code in a specific programming language to implement the solution methodology, testing, debugging, maintaining the code, verifying and validating the results, and monitoring the consumption of computer resources over the entire process. The objective of programming, therefore, is to develop a series of instructions that can automate the performance of a specific task, or to solve some specific problem formulation in a timely and efficient manner.

1.2 Relevance of Computer Programming to Engineers

Computer programming has become an increasingly advantageous and necessary skill for engineers and scientists competing in the global economy. By writing computer programs to automate tedious and repetitive tasks such as design calculations, or preparing, processing and analyzing large amounts of data which would otherwise be done by hand, engineers and scientists can drastically increase their productivity and efficiency. Competence in computer programming predisposes engineers and scientists to pursue and develop more creative and innovative solutions than their peers. Knowledge, and some level of experience in computer programming enables engineers and scientists to communicate more effectively with full-time programmers and information technology professionals they may work with and collaborate with on various projects.

1.3 Python

Python is a general-purpose programming language. *Python* can be used to perform complex mathematical and engineering calculations, and to handle big data. *Python* can be used for building desktop applications and web apps. *Python* code is executed rapidly which enables quick prototyping or production-ready software development.

Python has a simpler syntax compared to some other popular programming languages. The simpler syntax enables programs to be written with fewer lines of code. Due to its simpler syntax and ease of use, *Python* is amenable for use by beginners as well as seasoned programmers. *Python* is increasingly popular as a first programming language for beginners.

As a result of its user-friendliness and versatility, *Python* is the programming language driving some of the internet's most popular websites, namely:

- *Google*
- *Youtube*
- *Quora*
- *Dropbox*
- *Yahoo!*
- *Yahoo Maps*
- *Reddit*
- *Bitly*
- *Instagram*
- *Spotify*
- *SurveyMonkey*
- *Pintrest*
- *Eventbrite*
- *Firefox*
- *and many others*

2. CONDITIONAL STATEMENTS

2.1 Definition

A conditional statement is a basic feature of a programming language that executes different instructions (lines of code) based on whether some **condition** is met. Conditional statements enable the programmer to control the way an application interacts with the user by controlling the direction of flow of the code execution. Conditional statements are often referred to as **branching**, as they provide a means for a program to branch off in some direction or the other as some condition(s) is checked for and met, and the program then proceeds in the relevant direction(s).

2.2 The *if* Statement

The simplest conditional statement is the *if* statement. If a specified condition is met, a **block** of code will be executed. A block also called a **compound statement**, is simply a group of statements.

In *Python*, the *if* statement is of the structure:

```
if < conditional expression > :
…       < block >
```

The "…" represent an **indentation (**or an **indent)**. Unlike other programming languages, *Python* requires the block(s) to be indented away from the alignment with the *if* keyword. The indenting is accomplished by hitting the **Tab** key on the keyboard. During coding, to exit the block and hence the *if* statement, one must hit the **Enter** key to go to the next line, followed by hitting the **Backspace** key to realign the cursor with the *if* keyword.

The colon (" : ") is required after the conditional expression. Unlike other programs, in *Python*, the conditional expression does not have to be enclosed in parentheses. Some programming languages and scripting languages require the block to be enclosed in some brackets such as parentheses, curly brackets etc. This is not the case with *Python*.

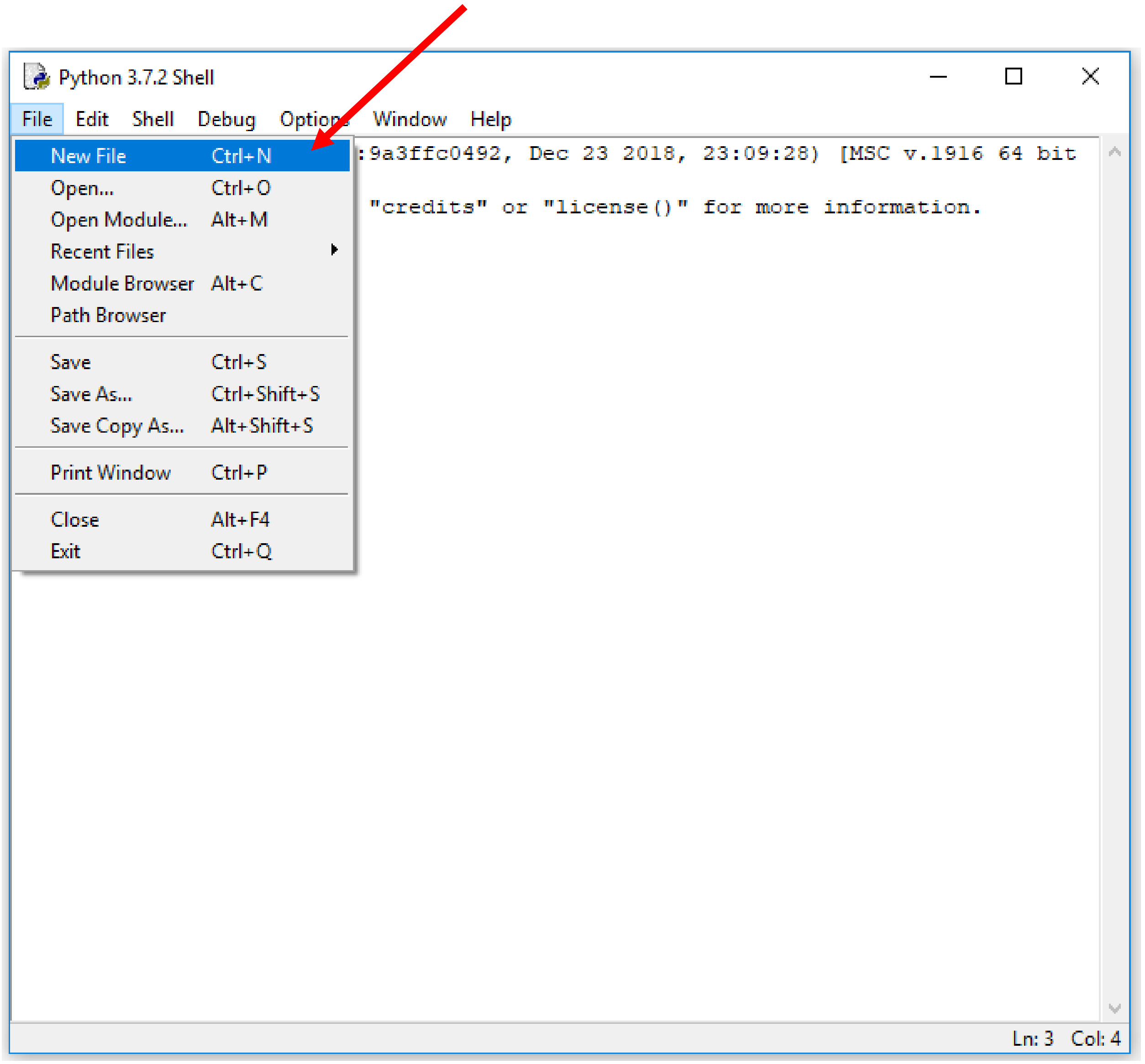
The *conditional expression* is a **logical expression** where a *Python* **logical operator** or comparison operator is applied to compare, evaluate, or check that the inputs (**operands**) meet the specified condition and return the Boolean result of "true", based upon which the block will then be executed. If the logical expression returns a value of "false", the block will not be executed and the cursor skips to the next line of code. If the next line of code is less indented than the block, then the cursor has transitioned out of the *if* statement.

The *Python* logical operators are summarized in Table 2.1.

Table 2. 1: Logical operators

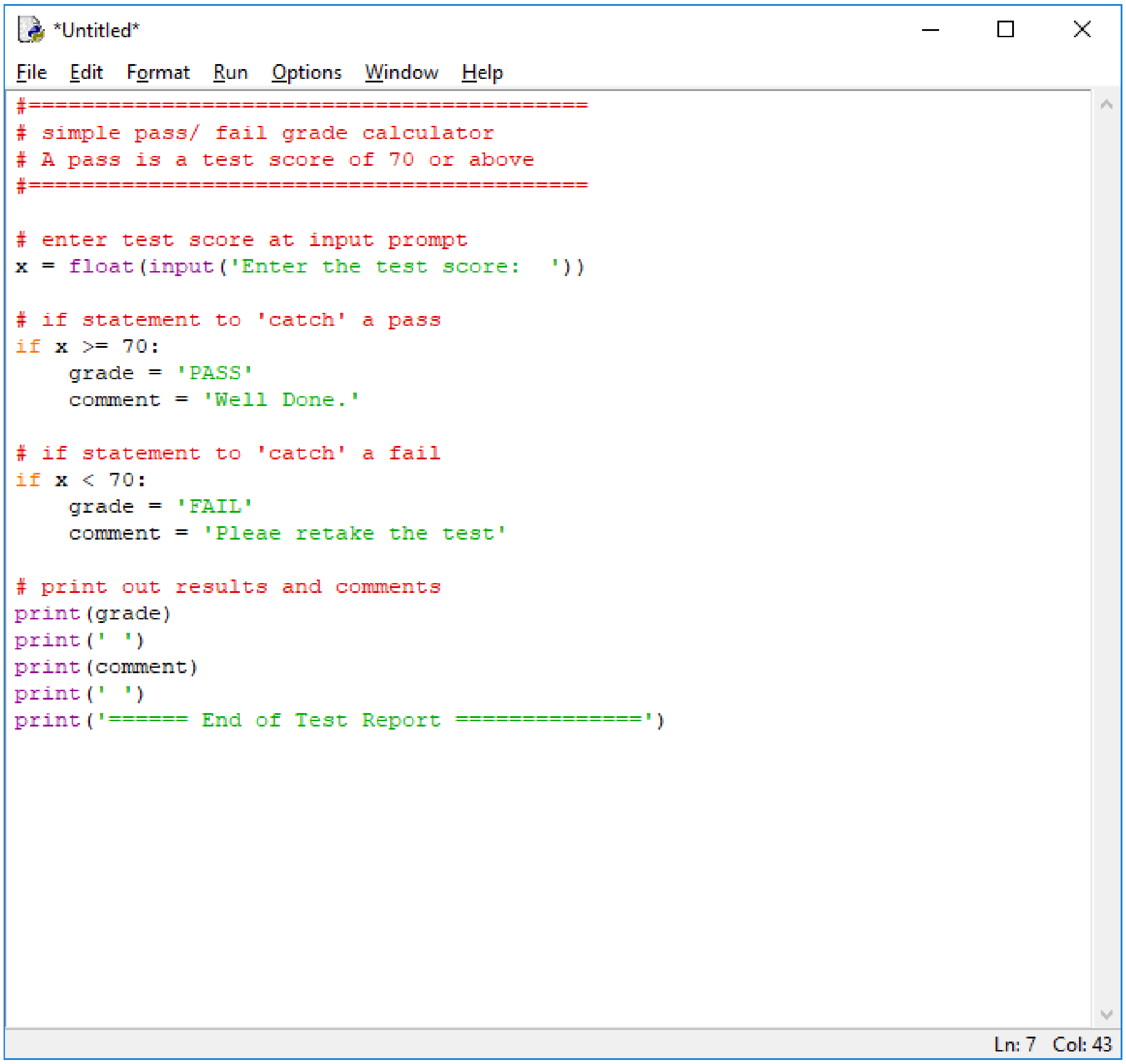
Operator	**Name**	**Example**	**Description**
==	Equals to	x==y	returns "true" if value of x equals value of y, otherwise returns "false"
!=	Not equal to	x!=y	returns "true" if value of x does not equal value of y, otherwise returns "false"
<	Less than	x<y	returns "true" if value of x is less than value of y, otherwise returns "false"
<=	Less than or equal to	x<=y	returns "true" if value of x is less than or equal to value of y, otherwise returns "false"
>	Greater than	x>y	returns "true" if value of x is greater than value of y, otherwise returns "false"
>=	Greater than or equal to	x>=y	returns "true" if value of x is greater than or equal to value of y, otherwise returns "false"

Open a new session of IDLE (Python GUI).
Click on **File**.
Click on **New File**, to open the File Editor.



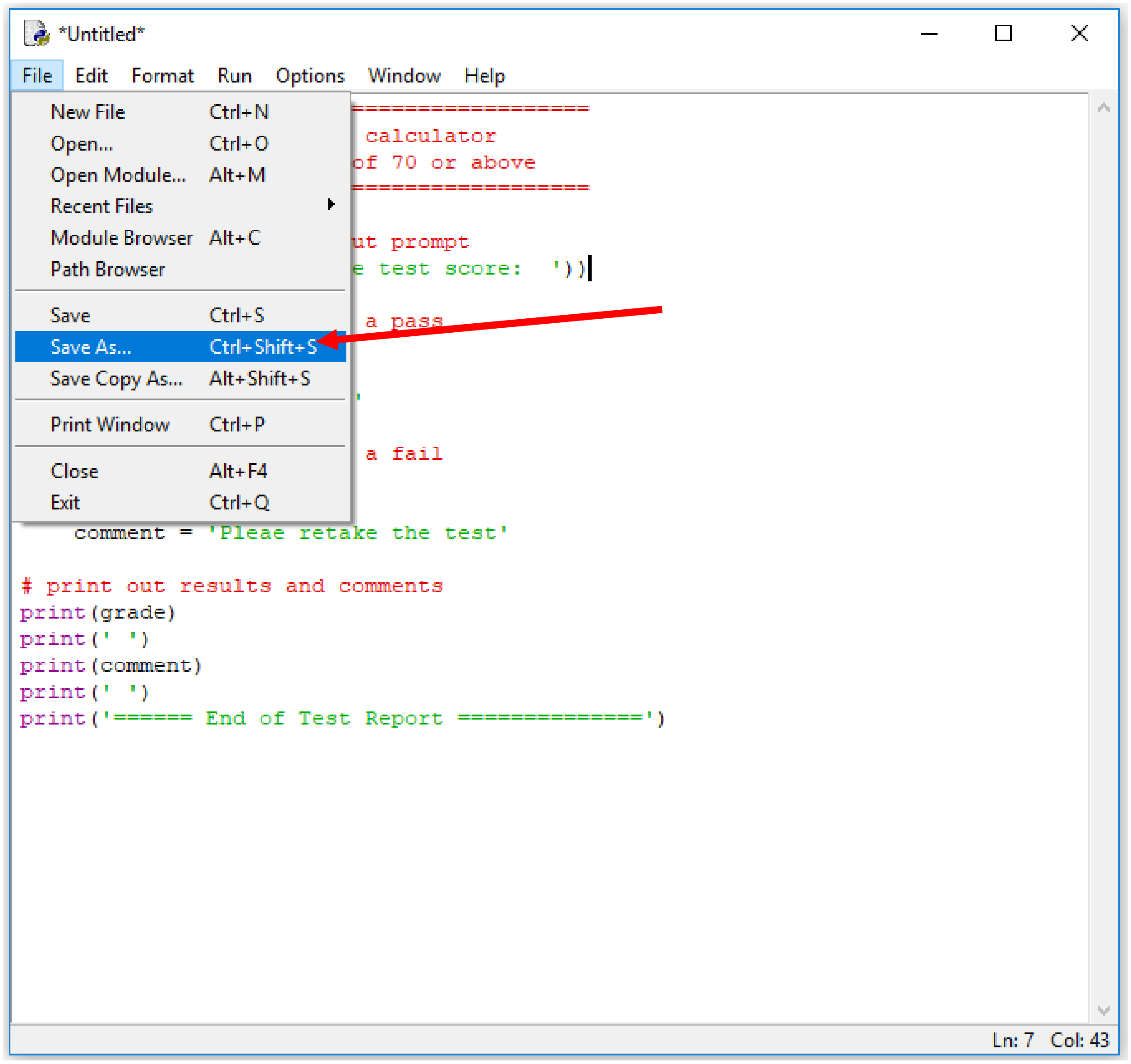
```
:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit

 "credits" or "license()" for more information.
```

In the File Editor type the following.



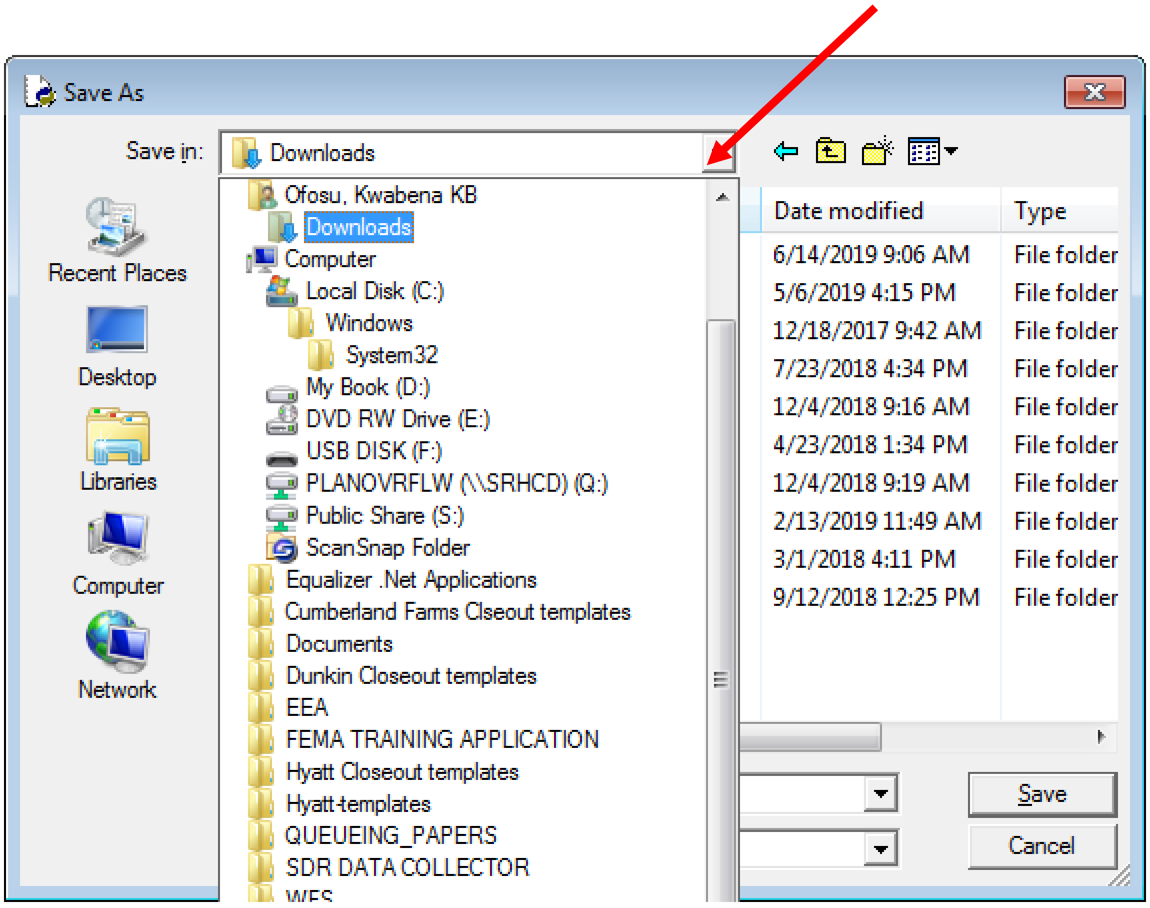
```
#===========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#===========================================

# enter test score at input prompt
x = float(input('Enter the test score:  '))

# if statement to 'catch' a pass
if x >= 70:
    grade = 'PASS'
    comment = 'Well Done.'

# if statement to 'catch' a fail
if x < 70:
    grade = 'FAIL'
    comment = 'Pleae retake the test'

# print out results and comments
print(grade)
print(' ')
print(comment)
print(' ')
print('====== End of Test Report ==============')
```

Save the file.



```
==================
 calculator
of 70 or above
==================

ut prompt
e test score:  '))

 a pass

'

 a fail

    comment = 'Pleae retake the test'

# print out results and comments
print(grade)
print(' ')
print(comment)
print(' ')
print('====== End of Test Report ==============')
```

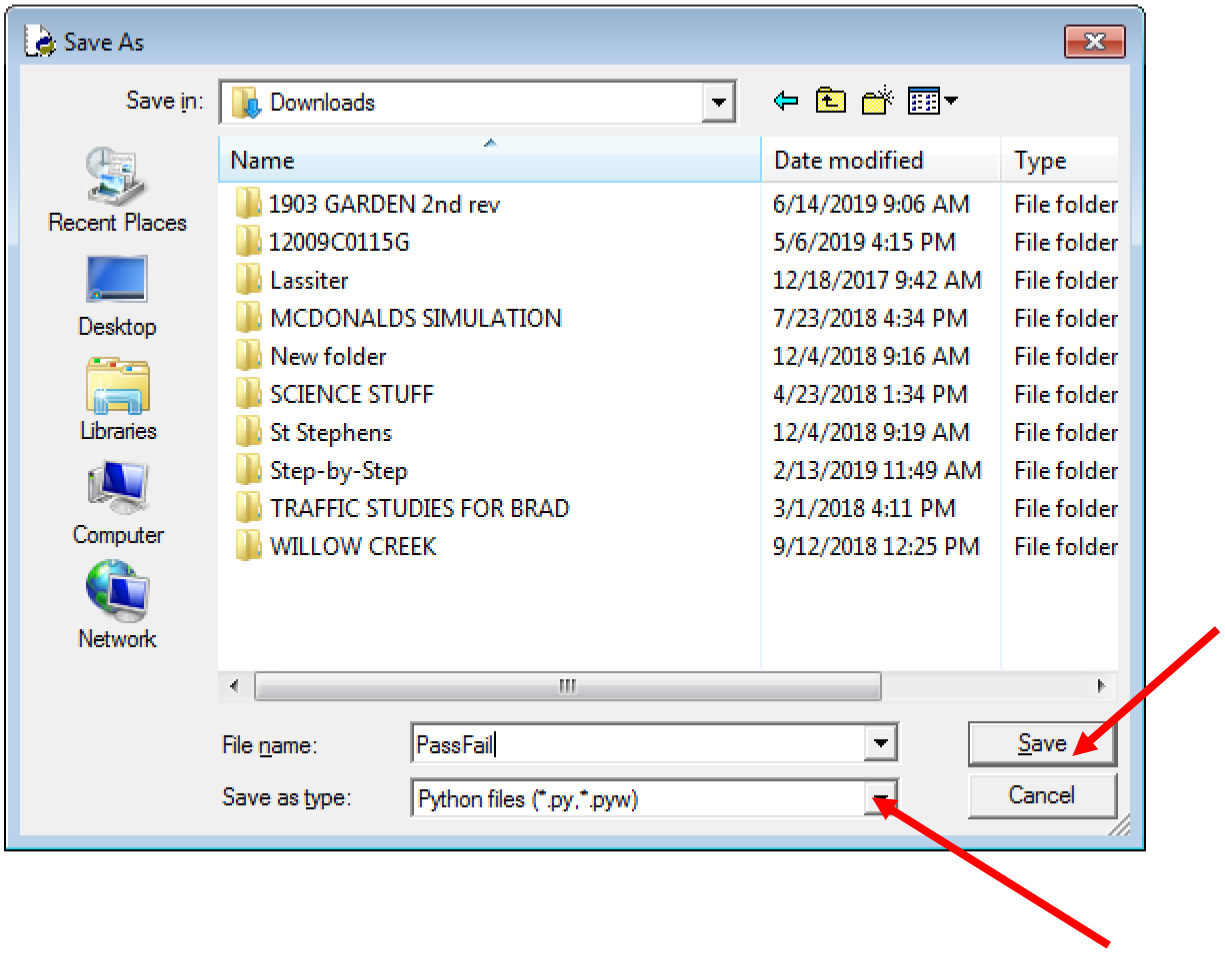
Navigate to a folder of your choice.



Date modified	Type
6/14/2019 9:06 AM	File folder
5/6/2019 4:15 PM	File folder
12/18/2017 9:42 AM	File folder
7/23/2018 4:34 PM	File folder
12/4/2018 9:16 AM	File folder
4/23/2018 1:34 PM	File folder
12/4/2018 9:19 AM	File folder
2/13/2019 11:49 AM	File folder
3/1/2018 4:11 PM	File folder
9/12/2018 12:25 PM	File folder

with a

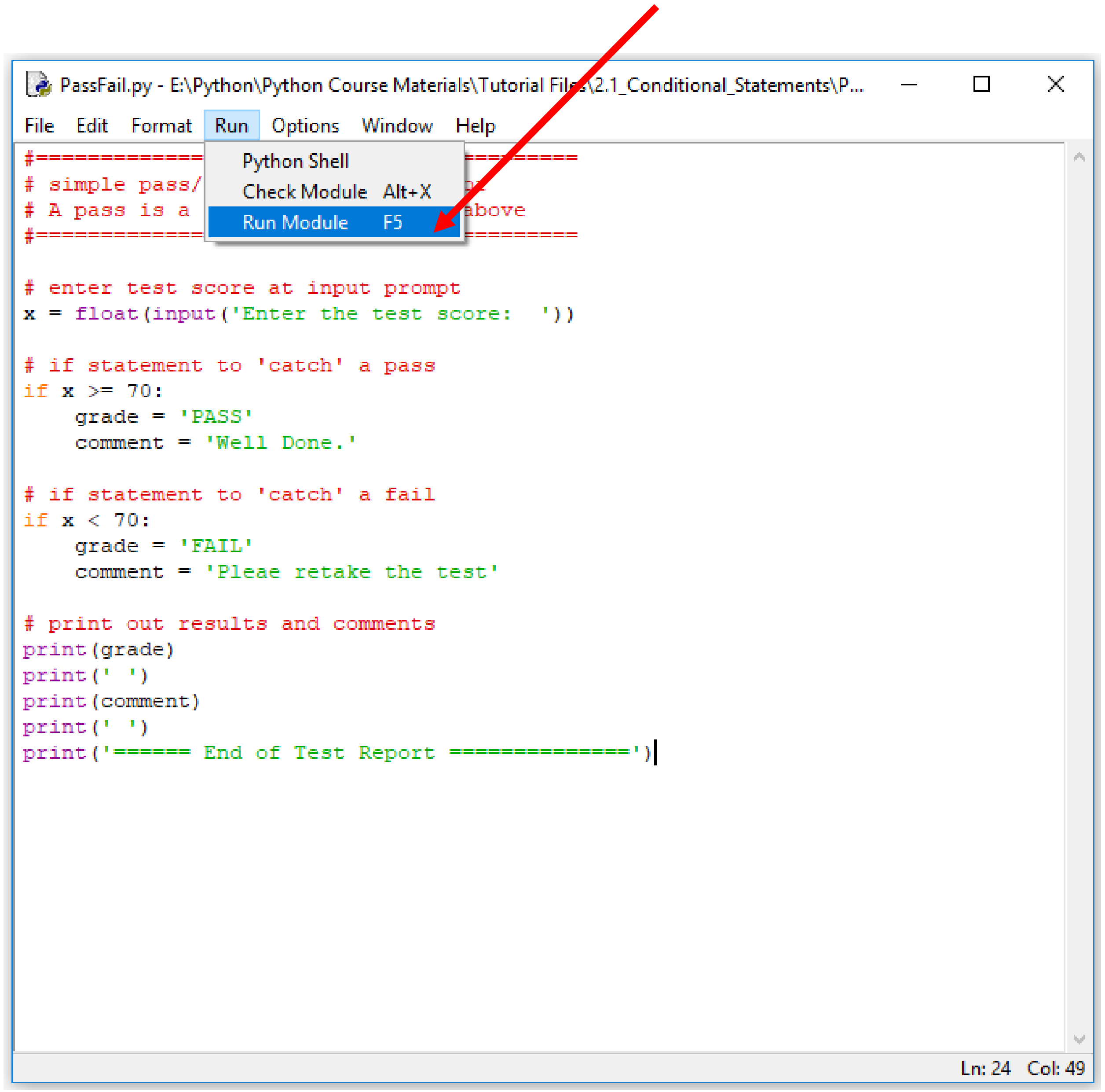
Give your file a name of your choice.
Select the Python suffix for the file type.
Hit **Save** to save the file.



Name	Date modified	Type
1903 GARDEN 2nd rev	6/14/2019 9:06 AM	File folder
12009C0115G	5/6/2019 4:15 PM	File folder
Lassiter	12/18/2017 9:42 AM	File folder
MCDONALDS SIMULATION	7/23/2018 4:34 PM	File folder
New folder	12/4/2018 9:16 AM	File folder
SCIENCE STUFF	4/23/2018 1:34 PM	File folder
St Stephens	12/4/2018 9:19 AM	File folder
Step-by-Step	2/13/2019 11:49 AM	File folder
TRAFFIC STUDIES FOR BRAD	3/1/2018 4:11 PM	File folder
WILLOW CREEK	9/12/2018 12:25 PM	File folder

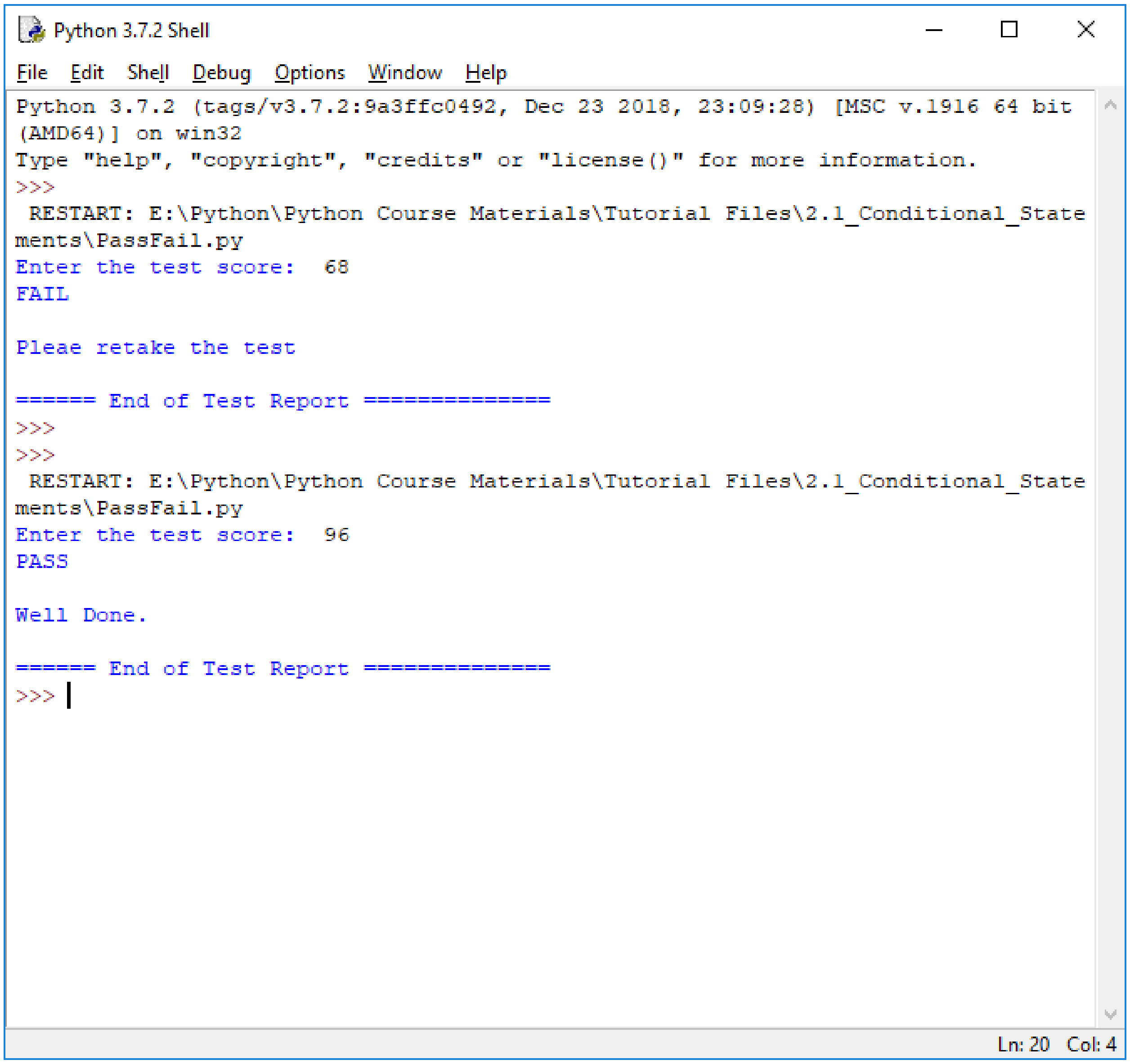
File name: PassFail
Save as type: Python files (*.py,*.pyw)

In the File Editor
Click on **Run**.
Click on **Run Module**, to run the file.


PassFail.py - E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_Statements\P...
File
Edit
Format
Run
Options
Window
Help
Python Shell
Check Module Alt+X
Run Module F5
Ln: 24 Col: 49

```
#=============                          ==========
# simple pass/                          e
# A pass is a                           above
#=============                          ==========

# enter test score at input prompt
x = float(input('Enter the test score:  '))

# if statement to 'catch' a pass
if x >= 70:
    grade = 'PASS'
    comment = 'Well Done.'

# if statement to 'catch' a fail
if x < 70:
    grade = 'FAIL'
    comment = 'Pleae retake the test'

# print out results and comments
print(grade)
print(' ')
print(comment)
print(' ')
print('====== End of Test Report ==============')
```

Follow the prompts to complete the program execution in IDLE.
Review the results in IDLE.



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\PassFail.py
Enter the test score:  68
FAIL

Pleae retake the test

====== End of Test Report ==============
>>>
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\PassFail.py
Enter the test score:  96
PASS

Well Done.

====== End of Test Report ==============
>>> 
```

2.3 The *if…else* Statement

The *if…else* statement is used to specify a block of code to run if the condition is met, and another block of code to run if the condition is not met. The syntax is as follows.

```
if < conditional expression > :
…       < block >
else :
…       < block >
```

2.4 The *if…elif…else* Statement

In the *if…elif…else* statement, the branching execution is based on several alternatives using as many *elif* (short for *else if*) statements as needed. *Python* evaluates the *elif* expressions in succession and executes the block associated with the first *elif* that evaluates to “true”, only, and then exits the *if…elif…else* statement altogether.

If none of the *elif* expressions evaluates to “true”, then the *else* block will be executed. However, as with other programming languages, the *else* condition is not required, and may be omitted, in which case no block will be executed if none of the *elif* conditions evaluates to “true”. The programmer must determine whether the *else* condition is necessary, applicable, or relevant to the specific context of the *if* statement.

The syntax for the *if…elif…else* statement is

```
if < conditional expression > :
…       < block >
elif < conditional expression >  :
…       < block >
elif < conditional expression >  :
…       < block >
elif < conditional expression >  :
…       < block >
elif < conditional expression >  :
…       < block >
:
:
:
else :
…       < block >
```

2.5 Short Hand Syntax

A short hand or one-line syntax may alternately be used for conditional statements.

For the simple *if* statement, the one-line syntax is as follows.

if < conditional expression > : < block >

For the simple *if…else* statement, the one-line syntax is:

< block > if < conditional expression > < block > else

Note that the in the *if…else* short hand, the colons are omitted.

The short hand of a three condition statement is

<block> if <conditional> <block> else <block> if <conditional> <block> else

However, it can be seen that for many conditions, a multiple condition statement using the one-line syntax will be complex and difficult to work with. The one-line syntax is most useful and beneficial for simpler conditional statements, and should restricted in their application as such.

2.6 The *pass* Statement

In the *Python* conditional statements, once an *if*, *elif* or *else* condition has been set up, there must be a block associated with it. In many programming situations however, there may be the need to "do nothing" if a condition evaluates to "true". Since an empty block or omitting the block is not permissible, the *pass* keyword is inserted. The *pass* statement is therefore effectively a placeholder used to enable the interpreter to continue after the relevant condition, by effectively doing nothing.

An example of the syntax is of the form

```
if < conditional expression > :
…       pass  # don't do anything
```

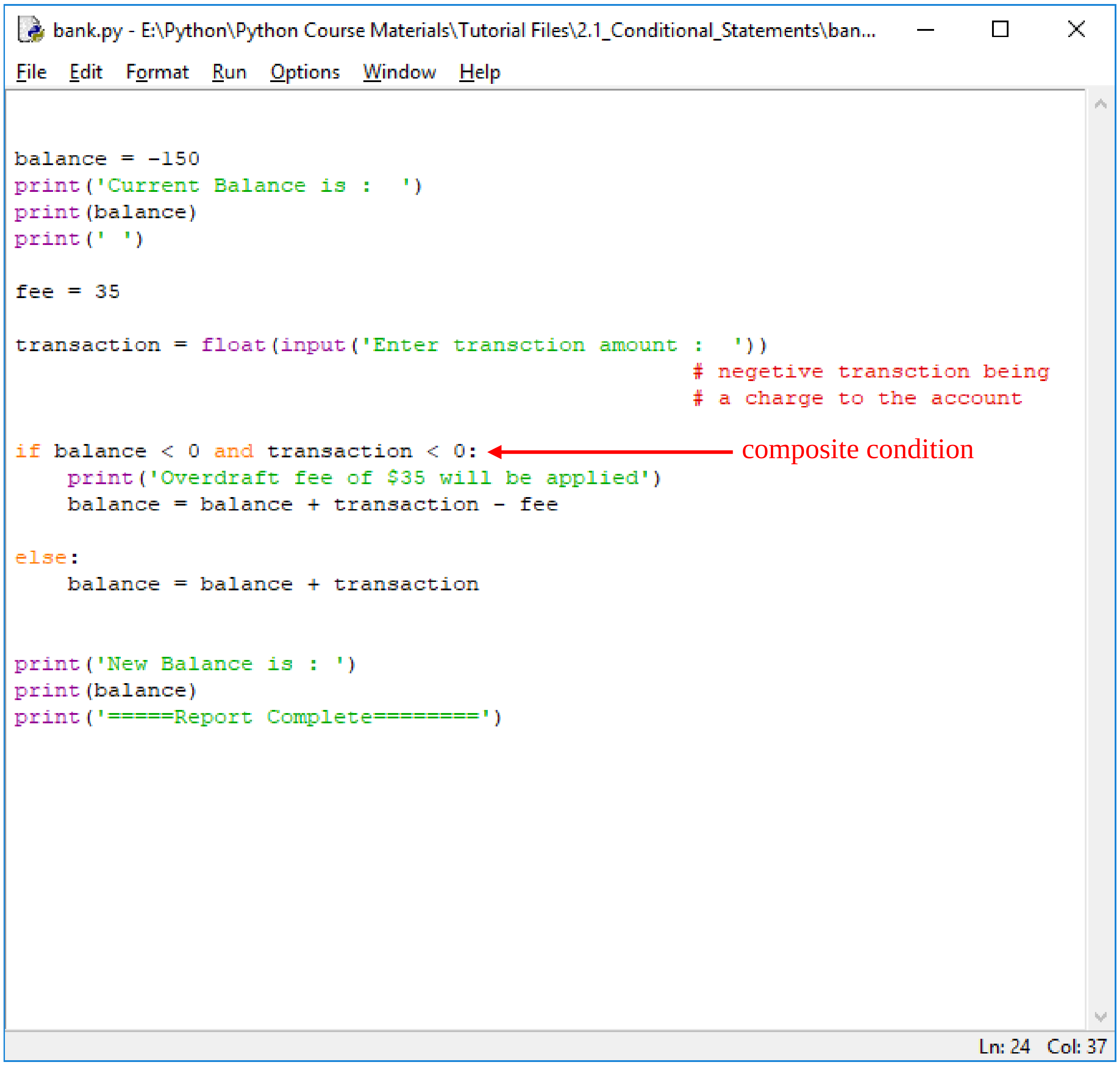
2.7 Composite Conditional Expressions

Conditional expressions may be combined using the **logical operators** such as "and", **"**or" and combinations thereof, to form a **composite conditional expression**.

For example, consider a bank account that is overdrawn. If another charge comes in and the bank pays it, the account goes further into the negative and is charged an overdraft fee for that transaction. However, if a deposit comes in that reduces the deficit, even though the account is still in the negative, the account is not charged the overdraft fee for that transaction.

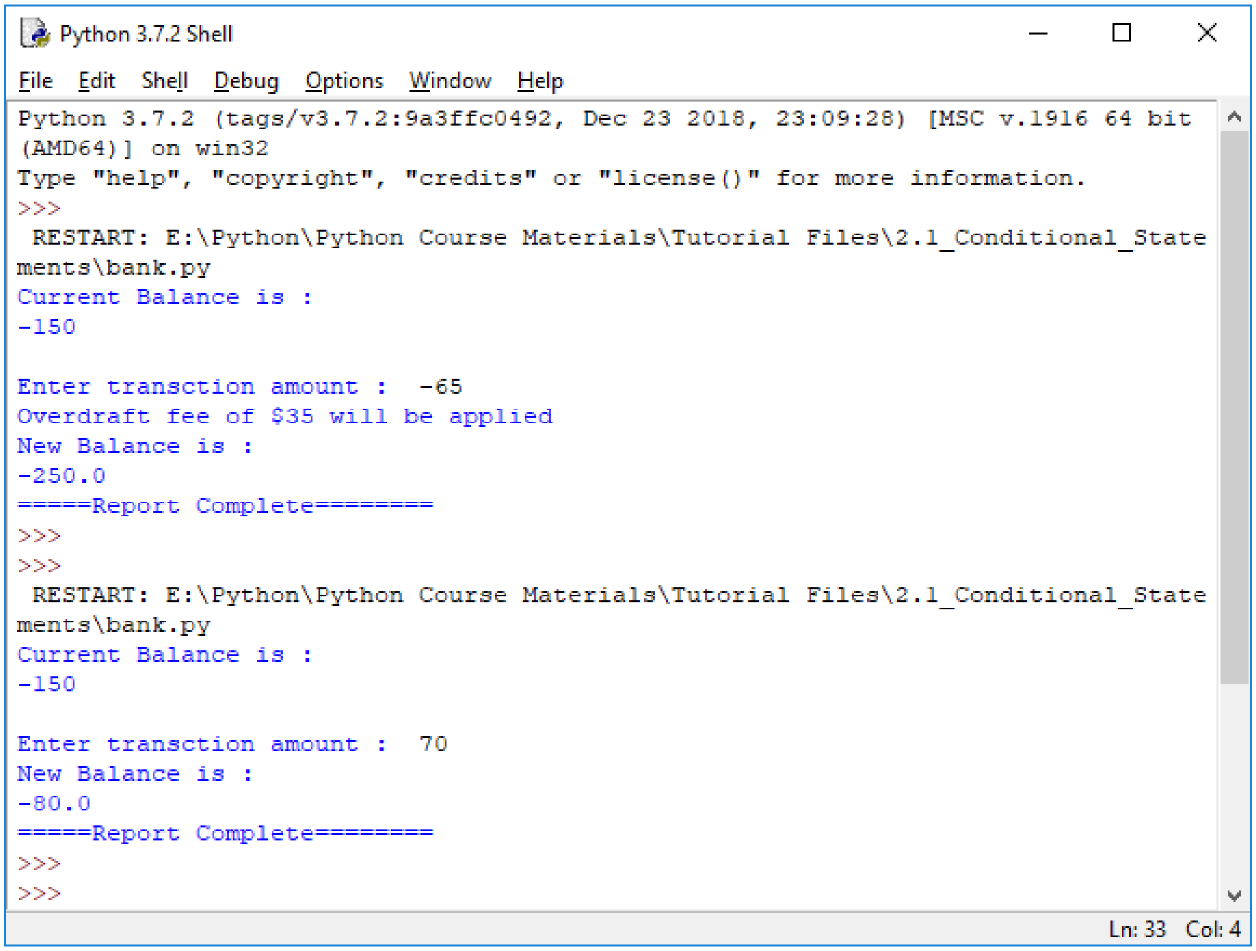
Using the negative sign for a charge transaction and positive sign for a deposit, the application of the overdraft fee can be programmed as shown below.

In the File Editor, reproduce the following.


bank.py - E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_Statements\ban...
File Edit Format Run Options Window Help
Ln: 24 Col: 37
composite condition

```
balance = -150
print('Current Balance is :  ')
print(balance)
print(' ')

fee = 35

transaction = float(input('Enter transction amount :  '))
                                                # negetive transction being
                                                # a charge to the account

if balance < 0 and transaction < 0:
    print('Overdraft fee of $35 will be applied')
    balance = balance + transaction - fee

else:
    balance = balance + transaction


print('New Balance is : ')
print(balance)
print('=====Report Complete========')
```

Run some trials of the program.
Review the results in IDLE.

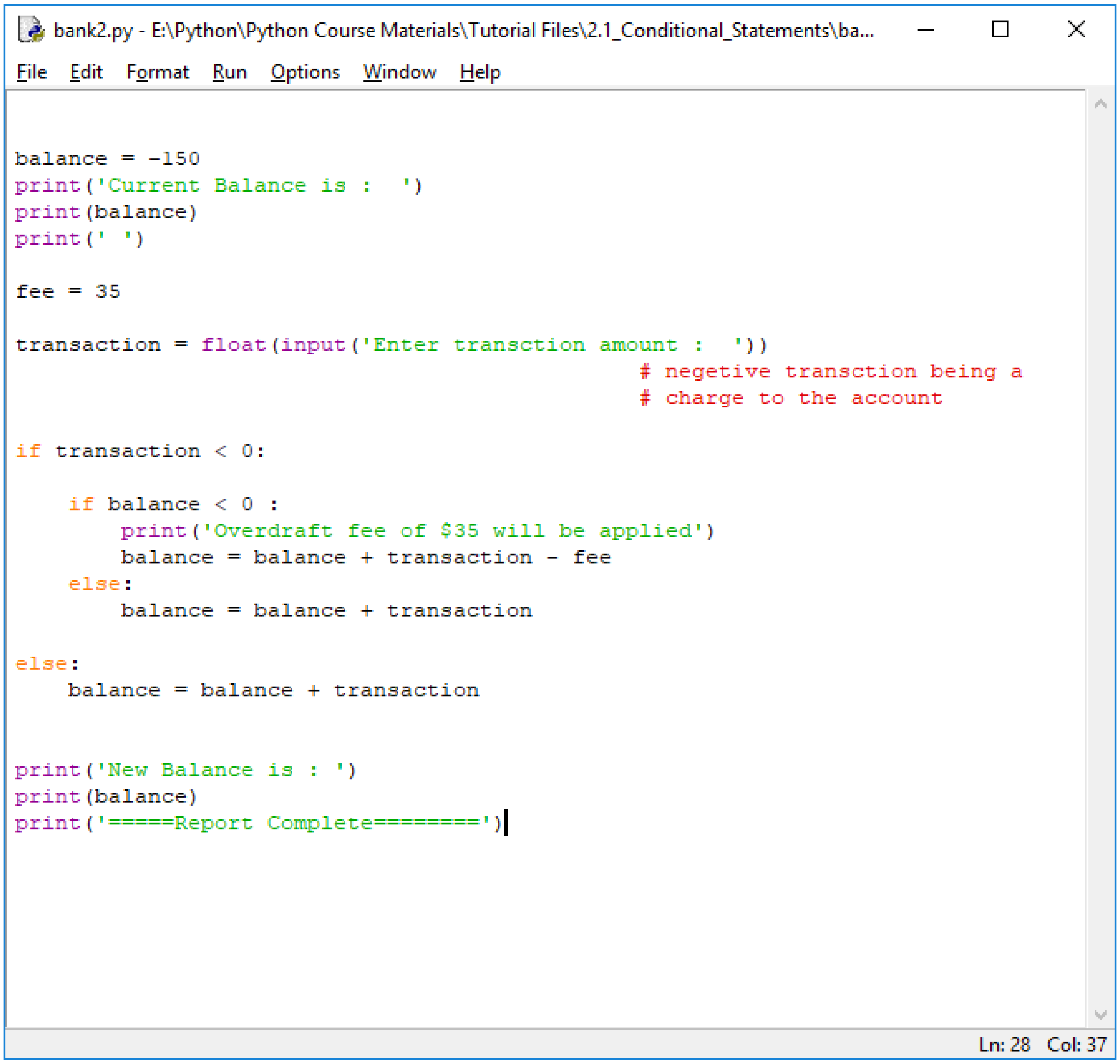


```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\bank.py
Current Balance is :
-150

Enter transction amount :  -65
Overdraft fee of $35 will be applied
New Balance is :
-250.0
=====Report Complete========
>>>
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\bank.py
Current Balance is :
-150

Enter transction amount :  70
New Balance is :
-80.0
=====Report Complete========
>>>
>>>
```

2.8 Nested Conditional Statements

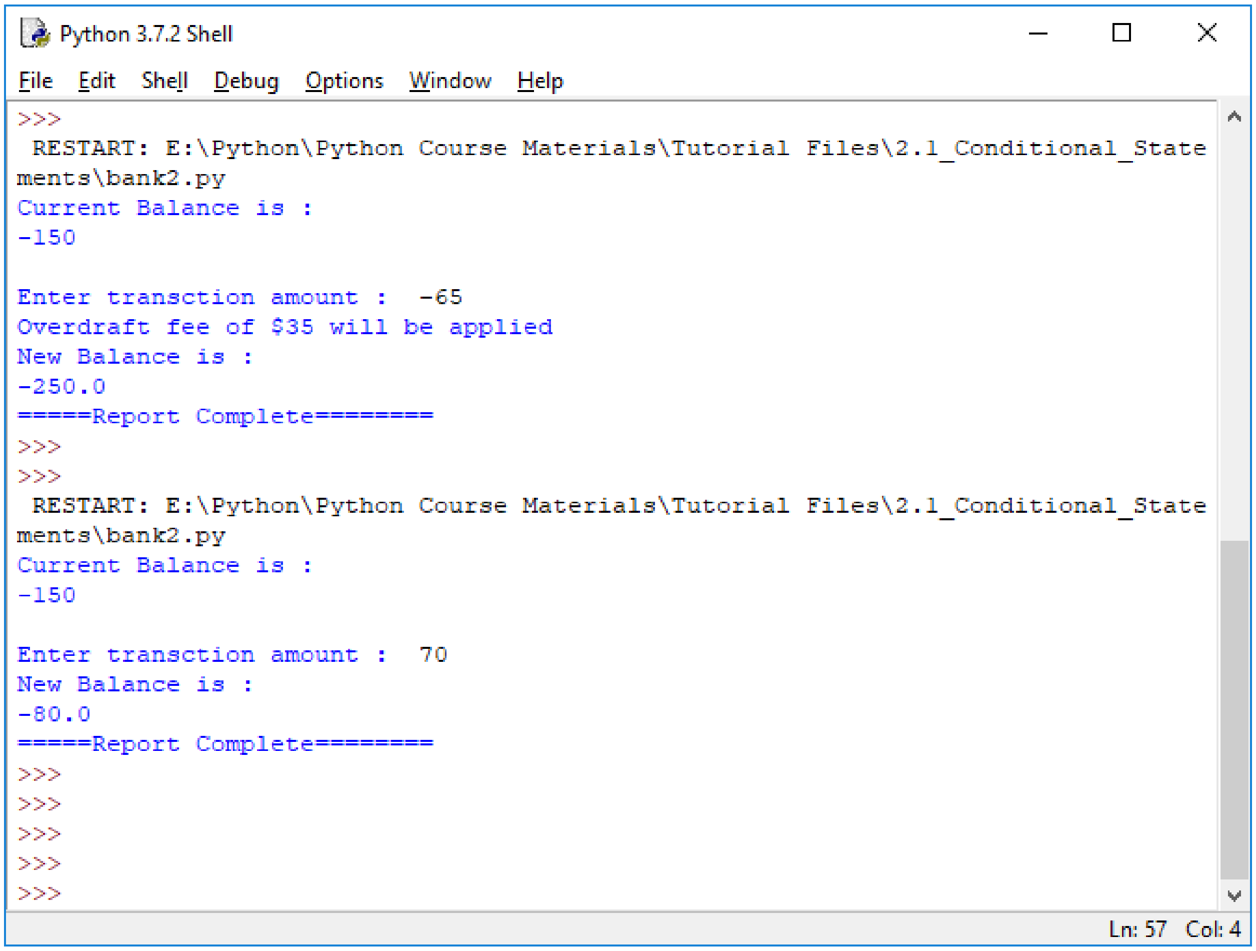
A **nested conditional statement** is a conditional statement placed inside another conditional statement. The bank account example can alternately be implemented using nested conditions as follows:

In the File Editor, reproduce the following.


bank2.py - E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_Statements\ba...
File Edit Format Run Options Window Help
Ln: 28 Col: 37

```
balance = -150
print('Current Balance is :  ')
print(balance)
print(' ')

fee = 35

transaction = float(input('Enter transction amount :  '))
                                                    # negetive transction being a
                                                    # charge to the account

if transaction < 0:

    if balance < 0 :
        print('Overdraft fee of $35 will be applied')
        balance = balance + transaction - fee
    else:
        balance = balance + transaction

else:
    balance = balance + transaction


print('New Balance is : ')
print(balance)
print('=====Report Complete========')
```

Run some trials of the program using the same numbers as with the composite conditional statement strategy.
Review the results in IDLE.
The results are identical.



```
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\bank2.py
Current Balance is :
-150

Enter transction amount :  -65
Overdraft fee of $35 will be applied
New Balance is :
-250.0
=====Report Complete========
>>>
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.1_Conditional_State
ments\bank2.py
Current Balance is :
-150

Enter transction amount :  70
New Balance is :
-80.0
=====Report Complete========
>>>
>>>
>>>
>>>
>>>
```

There is no limit on the amount of nesting or the extent of the use of composite conditions. Either method may be nested within the other without limit. The choice, relevance, or advantage of nesting versus composite conditions, as well as combinations thereof, must be determined by the programmer based on the specific objectives and requirements of the application.

3. LOOPING

3.1 Definition

Looping is a procedure in a programming language or scripting language that performs repetitive (or **iterative**) tasks. The loop is a sequence of instructions (a block) that is executed repeatedly while or until some condition is met or satisfied. Each repetition is called an **iteration**. Looping is fundamental to all programming and scripting languages. Like other languages, *Python* has several looping constructs.

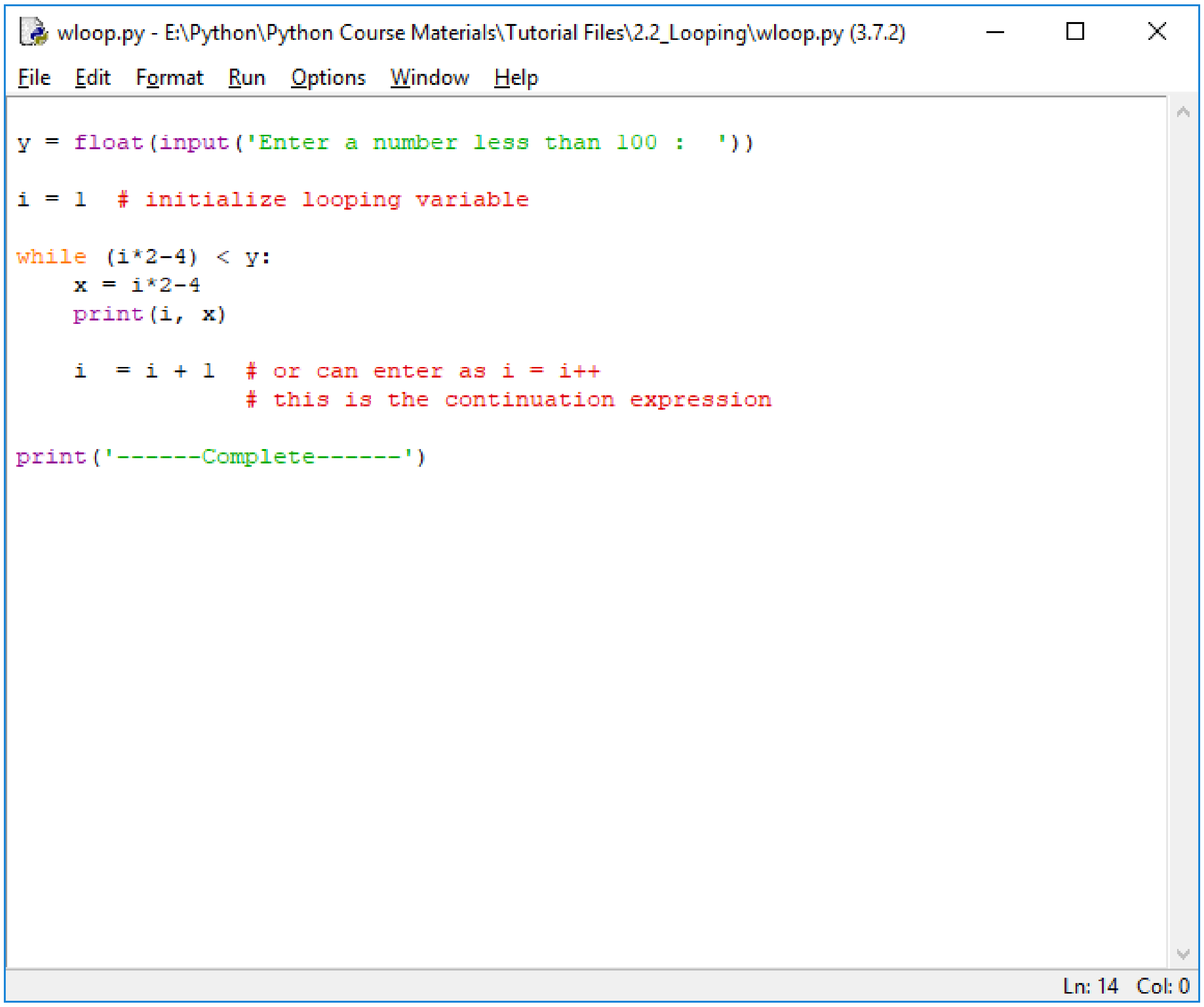
3.2 The *while* Loop

The while loop executes a block over and over again, indefinitely until some condition is met or satisfied. The syntax is:

```
while < expression > :
…       < block >
```

The expression controls the iterations of the loop. The looping expression typically involves a **looping variable**(s) and a logical expression. The looping expression establishes the condition(s) for which the loop will continue to run or shall be terminated. The looping variable is initialized prior to the start of the loop and is modified at some point in the block. At the end of an iteration, the looping variable will be updated (increased or decreased). The current value of the looping variable will now be checked against the looping expression; and if it evaluates to “true”, the next iteration will be executed, otherwise the loop will terminate.

In the File Editor, reproduce the following code.


wloop.py - E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\wloop.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 14 Col: 0

```
y = float(input('Enter a number less than 100 :  '))

i = 1  # initialize looping variable

while (i*2-4) < y:
    x = i*2-4
    print(i, x)

    i  = i + 1  # or can enter as i = i++
                # this is the continuation expression

print('------Complete------')
```

Let us go over the first few calculations manually.

1st Iteration:
We are asked to enter a number, let's say we enter 28.
So now $y = 28$
Next the looping variable is set to $i = 1$
In the looping expression calculate i*2 – 4 = 1*2 - 4 = -2 which is less than 28
So we shall run the loop.
$x = i$*2 – 4 = 1*2 - 4 = -2

print (*i*, *x*) which will display (1, -2)
And now we increment the looping variable $i = i + 1 = 1+1 = 2$
This completes the 1st iteration.

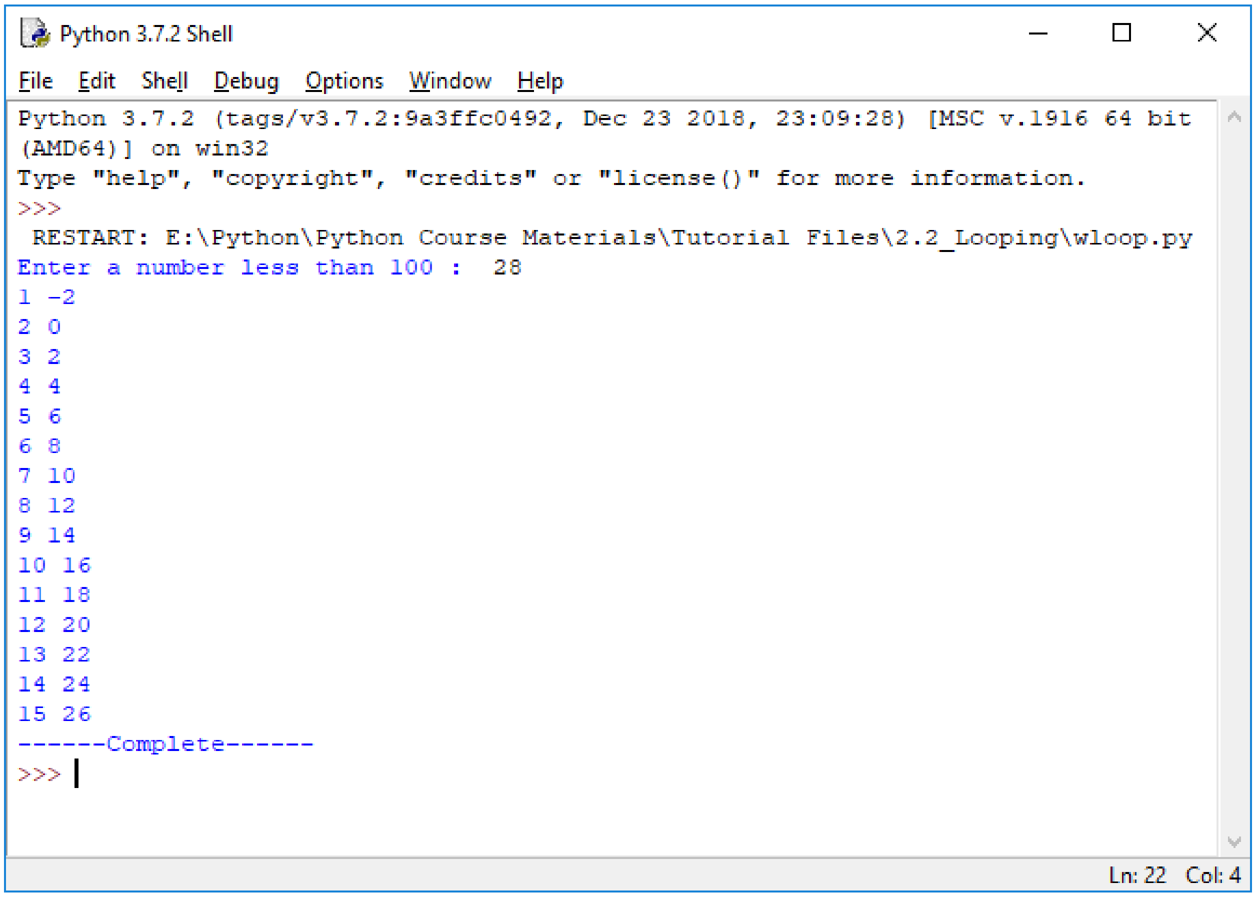
2nd Iteration:
$y = 28$
looping variable $i = 2$
looping expression $i*2 – 4 = 2*2 - 4 = 0$ which is less than 28
OK. We shall run the loop.
$x = i*2 – 4 = 2*2 - 4 = 0$
print (*i*, *x*) which will display (2, 0)
increment the looping variable $i = i + 1 = 1+1 = 2$

3rd Iteration:
$y = 28$
looping variable $i = 3$
looping expression $i*2 – 4 = 3*2 - 4 = 2$ which is less than 28
OK. We shall run the loop.
$x = i*2 – 4 = 3*2 - 4 = 3$
print (*i*, *x*) which will display (3, 2)

and so on and so forth, until,

16th Iteration:
$y = 28$
looping variable $i = 16$
looping expression $i*2 – 4 = 16*2 - 4 = 28$ which is NOT less than 28
The loop will now terminate immediately.
The last successful loop to run was for $i = 15$

The results in IDLE, are as follows.

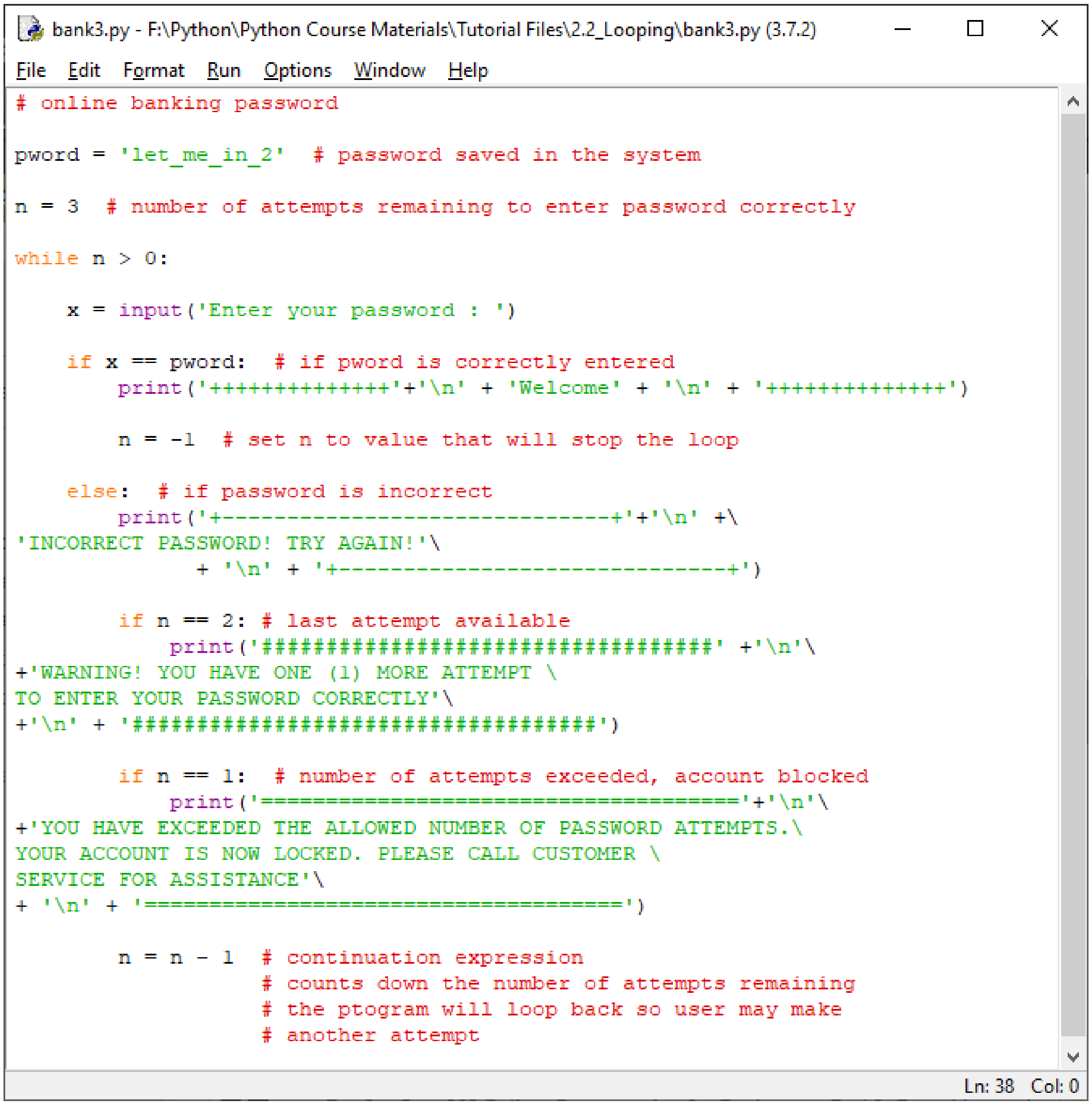


```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\wloop.py
Enter a number less than 100 :  28
1 -2
2 0
3 2
4 4
5 6
6 8
7 10
8 12
9 14
10 16
11 18
12 20
13 22
14 24
15 26
------Complete------
>>> |
```

It is pertinent to note that the increment/decrement of the looping variable does not have to be in steps of unity (1). If necessary, any integer value increment/decrement may be implemented by modifying the increment/decrement operation accordingly (e.g. *k++2* will give an increment of 2, etc.). Also, if applicable, the loop may be run "backwards", i.e., a higher initial value is assigned, and the continuation expression decrements the looping variable down towards the limit set in the looping expression.

For example: Consider your online banking account that gives you 3 attempts to correctly enter your password. As incorrect passwords are entered, the loop counts down to close the account.

In the File Editor, reproduce the following code.


bank3.py - F:\Python\Python Course Materials\Tutorial Files\2.2_Looping\bank3.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 38 Col: 0

```python
# online banking password

pword = 'let_me_in_2'  # password saved in the system

n = 3  # number of attempts remaining to enter password correctly

while n > 0:

    x = input('Enter your password : ')

    if x == pword:  # if pword is correctly entered
        print('++++++++++++++'+'\n' + 'Welcome' + '\n' + '++++++++++++++')

        n = -1  # set n to value that will stop the loop

    else:  # if password is incorrect
        print('+------------------------------+'+'\n' +\
'INCORRECT PASSWORD! TRY AGAIN!'\
              + '\n' + '+------------------------------+')

        if n == 2: # last attempt available
            print('###################################' +'\n'\
+'WARNING! YOU HAVE ONE (1) MORE ATTEMPT \
TO ENTER YOUR PASSWORD CORRECTLY'\
+'\n' + '###################################')

        if n == 1:  # number of attempts exceeded, account blocked
            print('====================================='+'\n'\
+'YOU HAVE EXCEEDED THE ALLOWED NUMBER OF PASSWORD ATTEMPTS.\
YOUR ACCOUNT IS NOW LOCKED. PLEASE CALL CUSTOMER \
SERVICE FOR ASSISTANCE'\
+ '\n' + '=====================================')

        n = n - 1  # continuation expression
                   # counts down the number of attempts remaining
                   # the ptogram will loop back so user may make
                   # another attempt
```

In this example, the following lexical features of *Python* are introduced:

'\n' : This is the **newline** character. It enables multiline output from the *print* command.

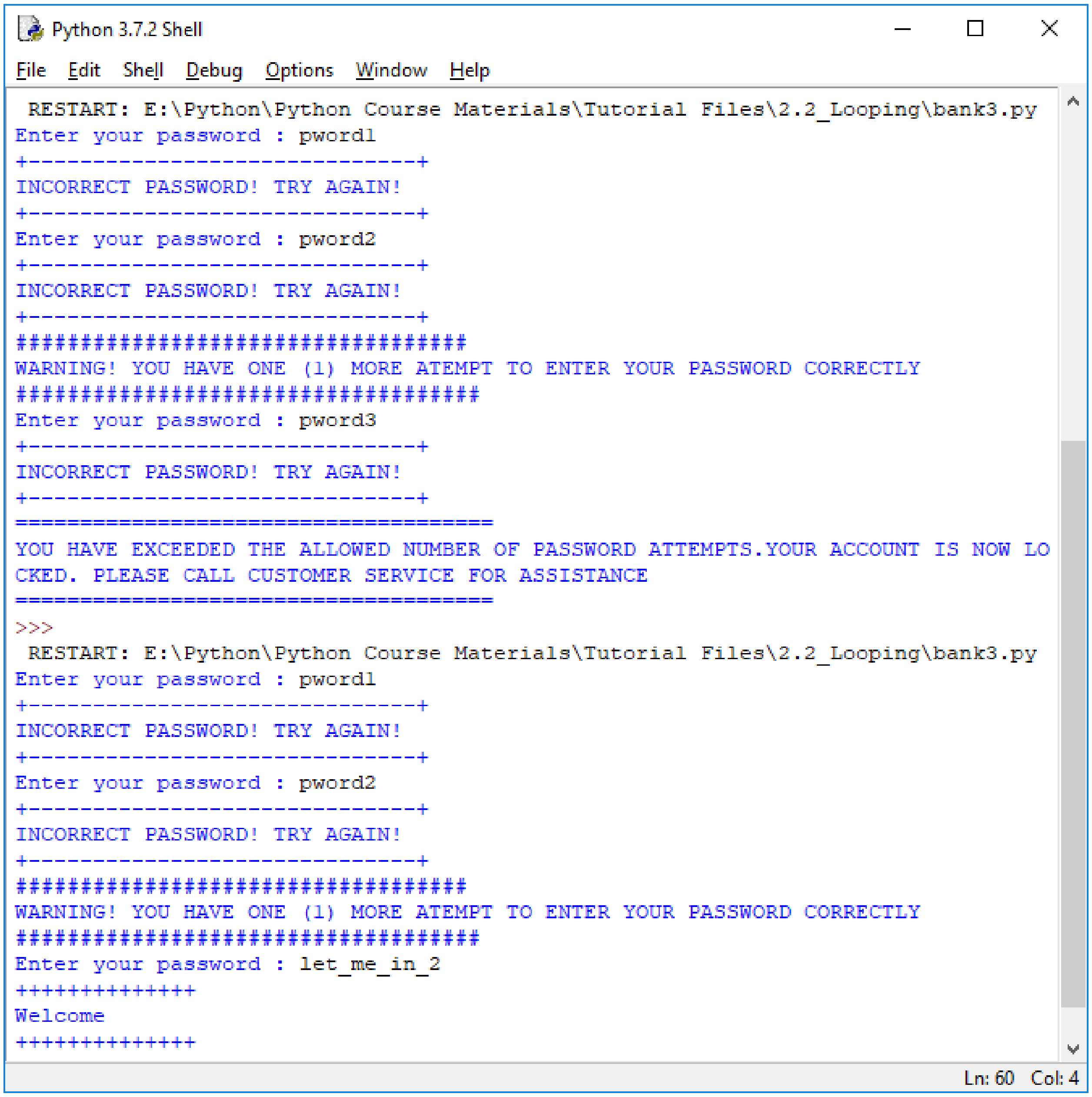
'\' : This is the **line break** (or **line continuation**) character. It is used to break a long line of code into manageable multiple lines for the visual benefit of the programmer only. The *Python* interpreter executes lines broken with the '\' character as one line of code. The requirement is that once used on a line there shall be nothing, not even white space, to the right of the '\' character.

n = n + 1 : The continuation statement. This is where the looping variable is updated – incremented by a value of one (1), for the loop to move on to the next iteration. As with other programming languages, *Python* supports the popular shorthand version *n = n++*. Therefore *n = n- -* will decrement the looping variable by a value of unity. *n = n++2* increments the looping variable by a value of two (2), whereas *n = n- -3* will decrement the looping variable by a value of three (3), and so on and so forth. For simplicity, in this course series the long hand version shall be used.

3.3 Nested Loops

The online banking password program uses nested loops and nested *if* statements. Loops of all types may be nested within each other as needed to achieve the desired functionality. All types of loops may be nested within all types of conditional statements and vice versa, without limit. The same applies for composite conditional statements. They may be incorporated into complex and intricate conditional statements and loops, and with nesting, as needed and without limit. It is the responsibility of the programmer to design and test the appropriate branching and looping structures that meet the needs and objectives of the project.

Test the online banking password code.



```
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\bank3.py
Enter your password : pword1
+--------------------------------+
INCORRECT PASSWORD! TRY AGAIN!
+--------------------------------+
Enter your password : pword2
+--------------------------------+
INCORRECT PASSWORD! TRY AGAIN!
+--------------------------------+
###################################
WARNING! YOU HAVE ONE (1) MORE ATEMPT TO ENTER YOUR PASSWORD CORRECTLY
####################################
Enter your password : pword3
+--------------------------------+
INCORRECT PASSWORD! TRY AGAIN!
+--------------------------------+
=====================================
YOU HAVE EXCEEDED THE ALLOWED NUMBER OF PASSWORD ATTEMPTS.YOUR ACCOUNT IS NOW LO
CKED. PLEASE CALL CUSTOMER SERVICE FOR ASSISTANCE
=====================================
>>> 
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\bank3.py
Enter your password : pword1
+--------------------------------+
INCORRECT PASSWORD! TRY AGAIN!
+--------------------------------+
Enter your password : pword2
+--------------------------------+
INCORRECT PASSWORD! TRY AGAIN!
+--------------------------------+
###################################
WARNING! YOU HAVE ONE (1) MORE ATEMPT TO ENTER YOUR PASSWORD CORRECTLY
####################################
Enter your password : let_me_in_2
++++++++++++++
Welcome
++++++++++++++
```

3.4 Infinite Loops

When using the *while* loop, there is always the danger of falling into an **infinite loop**. An infinite loop is a loop that lacks a functioning exit routine. As a result, the loop cannot stop, and repeats continuously until the operating system of your computer senses the issue and terminates the script, or until some event occurs, for instance having the script terminate automatically after a certain duration or number of iterations.

Recall our original *while* loop example.

```
y = input('Enter a number less than 100 :  ')

i = 1  # initialize looping variable

while (i*2 - 4) < y:
    x = i*2 - 4
    print(i, x)

    i  = i + 1  # or can enter as i = i++
                # this is the continuation expression

print('------Complete------')
```

Without actually doing it, to prevent potentially damaging your computer, consider what will happen if the looping expression is changed from

```
while (i*2 - 4) < y:
```

to

```
while (i*2 - 4) > -10:
```

Since the program starts at $i = 1$, the lowest value of the left-hand side of the expression is -2. So even though the expression is mathematically correct, it is mathematically impossible for the left-hand side to NOT be greater than -10. Thus, there will be no effective termination condition for this loop and the loop will run forever.

A related problem, though not technically an infinite loop, will also occur if you forget to implement the continuation expression, the increment (or decrement) of the looping variable. In that case, the first iteration will not be able to end, and the loop cannot move to the next iteration. The loop or the program is now stuck in the first iteration, forever.

Typically, an infinite loop will cause your interpreter, web browser, web page(s), and potentially your operating system, to crash, and may result in severe data loss. Therefore, *while* loops must be used with caution. The termination condition must be chosen carefully and studied closely. It is highly recommended that manual calculations be conducted for numerous scenarios to validate the math and the logic in order to ensure that an infinite loop will not occur. However, it must be mentioned that in spite of all these checks, even seasoned programmers fall into infinite loops every once in a while.

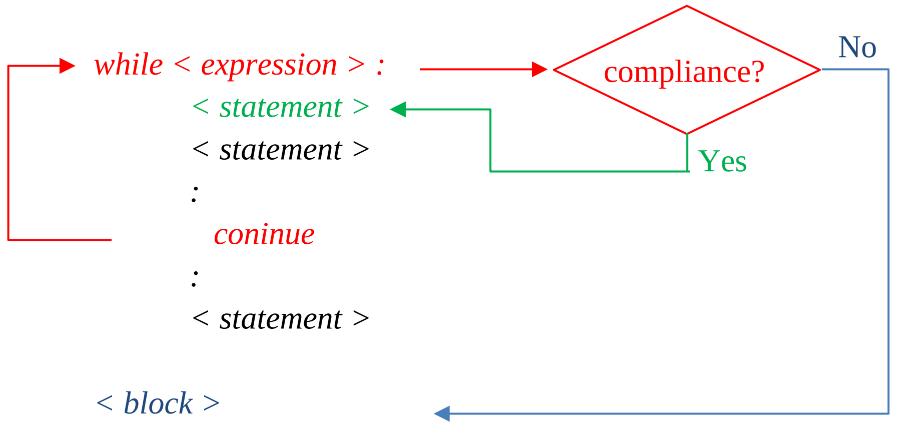
3.5 The *break* Statement

The *break* statement overrides the looping expression, the continuation expression and any applicable logic, and abruptly terminates the execution of the loop. The cursor jumps to the first line of code after (below) the body of the loop. The *break* statement can be depicted as follows.

```
while < expression > :
        < statement >
        < statement >
        :
          break ──────────┐
        :                 │
        < statement >     │
                          │
< block >      ◄──────────┘
```

3.6 The *continue* Statement

The *continue* statement abruptly stops the current iteration, cycles back to the looping expression for re-evaluation and if still valid continues with the execution of the next iteration. If the re-evaluation results in the looping expression being out of compliance, the entire loop terminates, and code execution continues at the first line of code after (below) the body of the loop. The *continue* statement can be depicted as follows.



```
while < expression > :
          < statement >
          < statement >
          :
            coninue
          :
          < statement >

< block >
```

3.7 The *else* Statement

The *else* statement in a *while* loop is of the form

```
while < expression > :
          < block >
else :
          < block >
```

The block under the *else* clause executes once the loop terminates. This is a unique feature of *Python* not found in other programming languages.

3.8 Short Hand Syntax

The *while* loop may alternately be implemented using a short hand or one-line syntax. However, as with the one-line *if* statements, it is recommended that they be restricted to the simpler applications. A one-line *while* loop will be of the form

```
< initialize loop variable >
while < looping expression > : < continuation statement >; < block >
```

3.9 The *for* Loop

The *Python for* loop iteratively executes statements on the items of a string, list, tuple, dictionary and set, hence forth collectively referred to as **iterables**. Unlike the *while* loop, the *for* loop does not require a looping variable to be set prior to execution.

The structure of the *Python* for loop is as follows.

```
for < var > in < iterable > :
…    < statement >
…    < statement >
…
```

where < *var* > denotes the loop variable which takes on the value of the next element of the iterable < *iterable* > each time through the loop. Indentation is required to separate the body of the loop from the *for* statement.

For professionals who have experience with other programming languages, it is pertinent to note that the *Python for* loop is significantly different from that of other programming languages.

As with the *while* loop, the *for* loop can incorporate the *break* statement, the *continue* statement and the *else* clause. Also, *for* loops can be nested in other branching and looping structures and vice versa as needed, with no restrictions as for as the programming is concerned.

The *for* loop structure on its own is not prone to infinite loops due to the fact that the looping is controlled by the number of items in the iterable. However, manipulating the variable in certain ways within the body of the loop, often done inadvertently, can indeed result in an infinite loop. As always, it is recommended to check, vet and recheck the code before running any kind of loop.

3.10 The *range( )* Function

The *range( )* function is used to loop through a block of code a specified number of times. The *range( )* function returns a sequence of numbers starting at zero (0) by default and ending at a specified number in increments of one (1) by default.

The syntax for the *range( )* function is of the form

```
for < variable > in range( < number > ) :
…       < block >
```

whereby the loop will run from a value of the variable equal to 0 through a value equal to the number specified minus one. For example:

```
for x in range( 8 ) :
…       < block >
```

This loop will run from *x* = 0 through *x* = 7 in increments of 1.

The starting value of the *range( )* function can be changed from the default value of 0 as follows.

```
for x in range( 2, 8 ) :
…       < block >
```

In this case the loop will run from *x* = 2 through *x* = 7 in increments of 1.

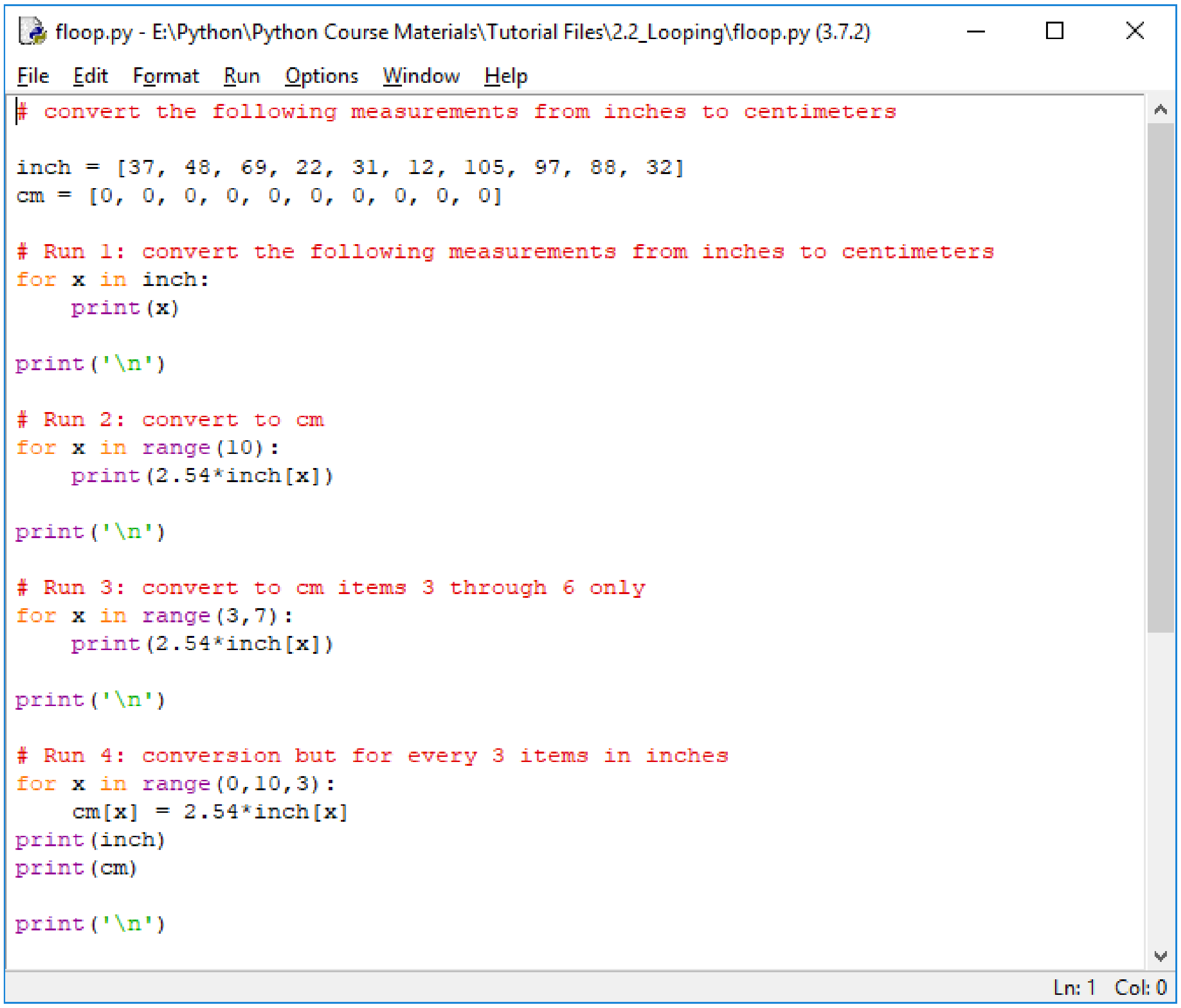
The size of the increment of the *range( )* function can be changed from the default value of 1 as follows:

for x in range(5, 12, 2) :
…   *< block >*

In this case the loop will run from *x* = 5 through *x* = 11 in increments of 2.

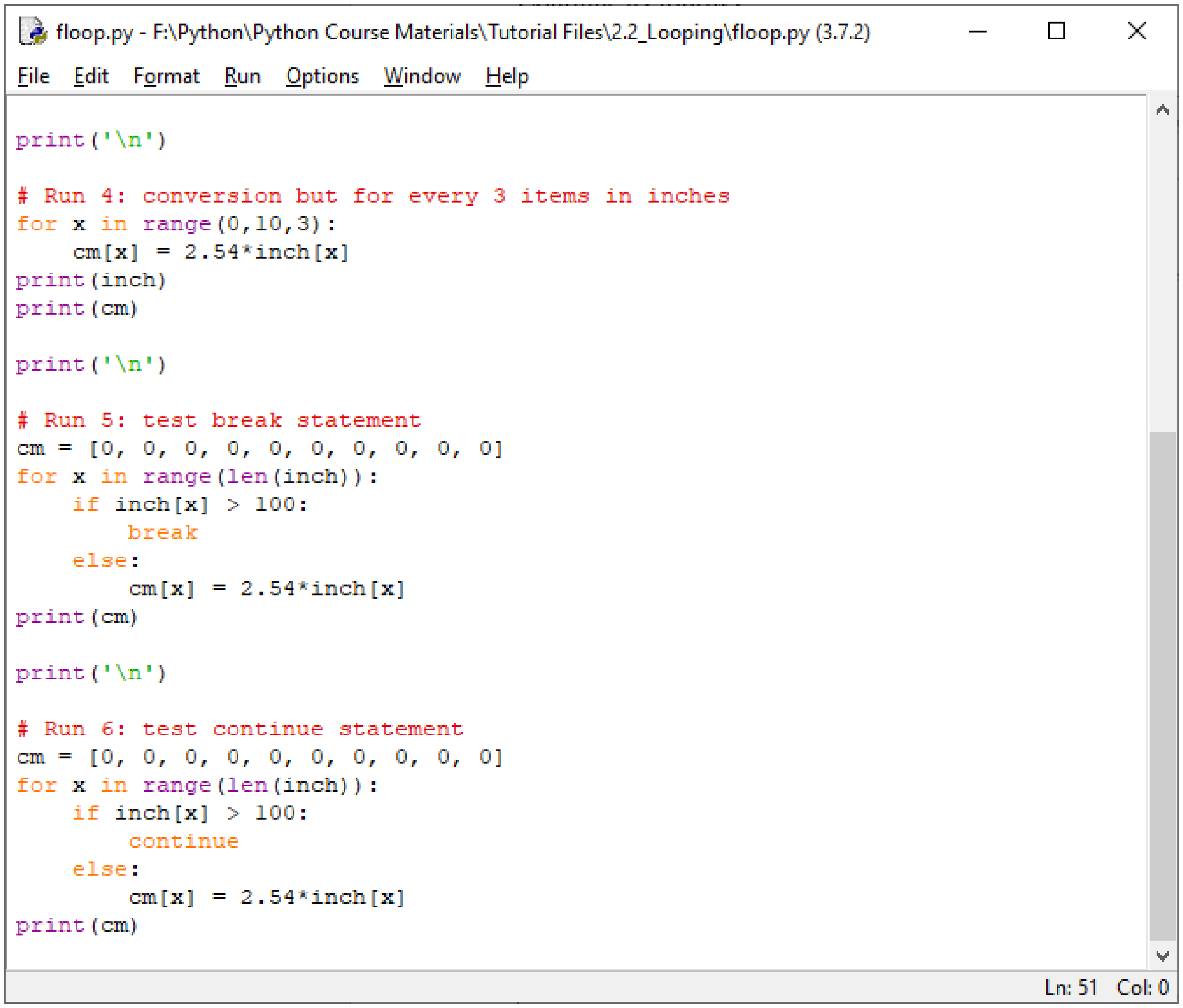
Professionals with some experience or familiarity with programming *for* loops in other programming languages will notice that incorporating the *range( )* function in the *for* loop enables the *Python for* loop to be applied in a similar manner as the *for* loop constructs in most other programming languages.

Open a new session of IDLE (Python GUI).
Open the File Editor.
Replicate the following.


floop.py - E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\floop.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
# convert the following measurements from inches to centimeters

inch = [37, 48, 69, 22, 31, 12, 105, 97, 88, 32]
cm = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

# Run 1: convert the following measurements from inches to centimeters
for x in inch:
    print(x)

print('\n')

# Run 2: convert to cm
for x in range(10):
    print(2.54*inch[x])

print('\n')

# Run 3: convert to cm items 3 through 6 only
for x in range(3,7):
    print(2.54*inch[x])

print('\n')

# Run 4: conversion but for every 3 items in inches
for x in range(0,10,3):
    cm[x] = 2.54*inch[x]
print(inch)
print(cm)

print('\n')
```

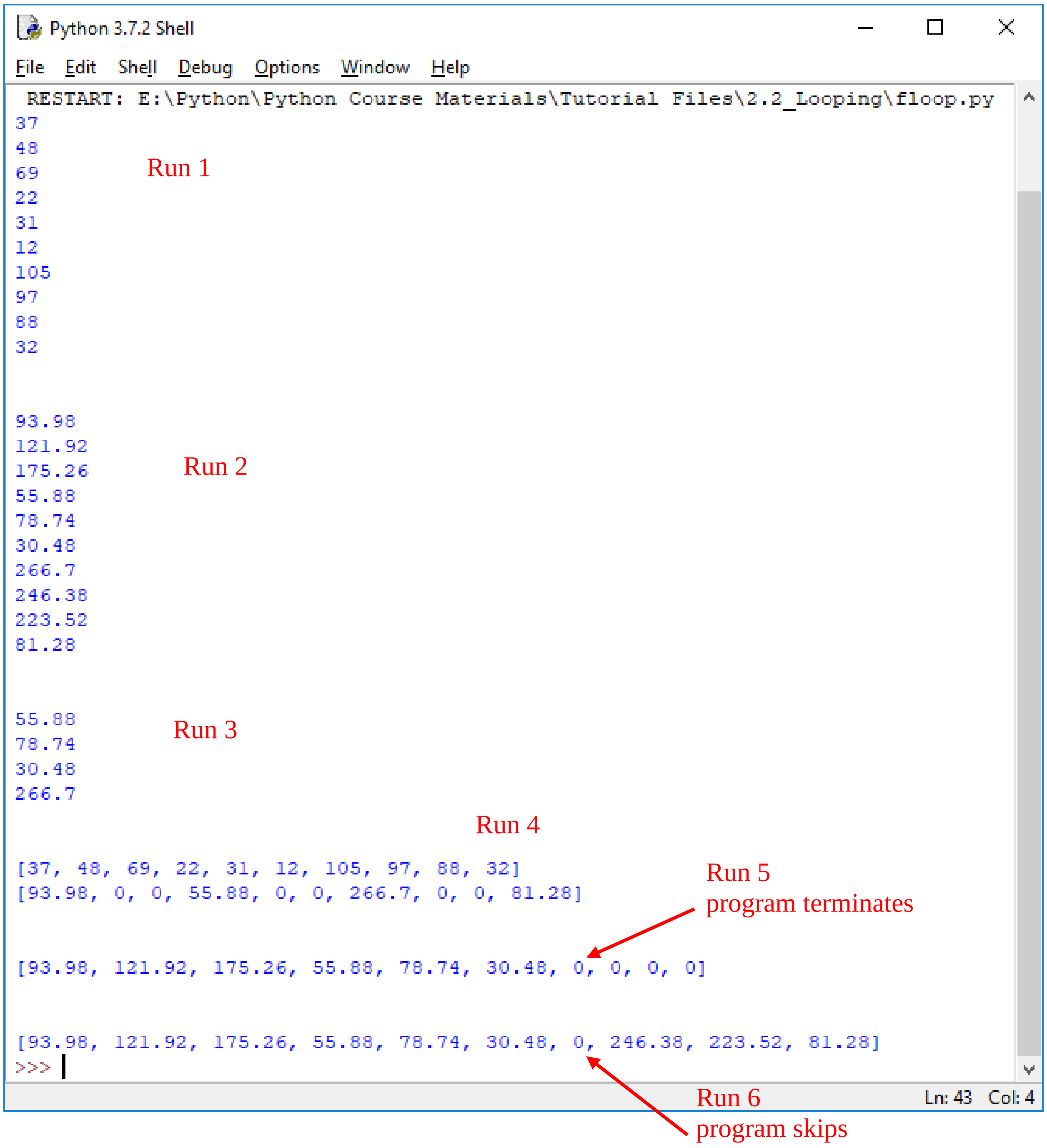
Continue as follows.


floop.py - F:\Python\Python Course Materials\Tutorial Files\2.2_Looping\floop.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 51 Col: 0

```
print('\n')

# Run 4: conversion but for every 3 items in inches
for x in range(0,10,3):
    cm[x] = 2.54*inch[x]
print(inch)
print(cm)

print('\n')

# Run 5: test break statement
cm = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for x in range(len(inch)):
    if inch[x] > 100:
        break
    else:
        cm[x] = 2.54*inch[x]
print(cm)

print('\n')

# Run 6: test continue statement
cm = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
for x in range(len(inch)):
    if inch[x] > 100:
        continue
    else:
        cm[x] = 2.54*inch[x]
print(cm)
```

Save the file.
Run the file.

Review the results in IDLE.



```
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.2_Looping\floop.py
37
48
69
22
31
12
105
97
88
32


93.98
121.92
175.26
55.88
78.74
30.48
266.7
246.38
223.52
81.28


55.88
78.74
30.48
266.7


[37, 48, 69, 22, 31, 12, 105, 97, 88, 32]
[93.98, 0, 0, 55.88, 0, 0, 266.7, 0, 0, 81.28]


[93.98, 121.92, 175.26, 55.88, 78.74, 30.48, 0, 0, 0, 0]


[93.98, 121.92, 175.26, 55.88, 78.74, 30.48, 0, 246.38, 223.52, 81.28]
>>> 
```

4. FUNCTIONS

4.1 Definition

In computing, **functions** and **procedures** are groups of instructions that perform a specific task. Generally, a function is a named group of instructions that conducts a specific task and can **return** a value. On the other hand, a procedure, also called a **routine**, performs a specific task but does not return a value. Functions and procedures, and variations thereof, are fundamental to all programming languages. Functions are therefore useful for conducting a specific task multiple times or repeatedly throughout a program by reusing a designated block of code.

A function runs only when it is **called**. A function may be set up such that some data is **passed** into it for it to execute. The data or **parameters** passed into a function are called **arguments**.

4.2 Creating a Function

In *Python*, a function is created by using the keyword *def*, as follows.

```
def  < function name > ( ) :
...        < block >
```

4.3 Calling a Function

To call a function, the name of the function is called followed by "empty" parentheses. For example:

```
def   Calculate_Salaries( )  :
...        < block >
:
< block >
Calcutate_Salaries( )
< block >
:
```

4.4 Arguments

Data may be passed into a function using arguments listed within the parentheses of the *def* line. Any number of arguments are permitted. Multiple arguments must be separated by commas. For example:

```
def   my_address(housenum, streetname, city, state )  :

…         print(housenum + ‘’ + streetname + ‘’ + city +‘’ +  state)
```

To call this function, the correct number of parameter values must be supplied in the correct positional order. This is referred to as **positional** or **required** arguments. For example:

```
 my_address(‘1186’, ‘Reading Drive’, ‘Hainesville’, ‘GA’ )
```

which produces the result

```
1186 Reading Drive Hainesville GA
```

An argument may be of any data type, such as a number, string, list, tuple, dictionary, etc. The argument will be treated in the same way as the data type is treated in the body of the function.

In *Python* arguments are **passed by reference**, which means what the argument is referring to can be changed within the body of the function. The opposite case, which is available in other programming languages, is to **pass by value**.

4.5 Default Arguments

A default argument is one that take on a default value if a value is not supplied when the function is called. For example:

```
 def   new_job(position = ‘Engineer’ )  :

…         print(‘Opening for ‘ +  position)
```

Calling the function,

```
 new_job(‘Programmer’ )
```

yields

```
Opening for Programmer
```

whereas calling

```
new_job( )
```

yields

```
Opening for Engineer
```

4.6 Keyword Arguments

With **keyword** (or **named**) arguments, the caller identifies the arguments by their specific name. This provides flexibility in that when the function is called, the arguments do not have to be passed in the specific positional order as in the function definition. The Python interpreter uses the keywords to match the values to the parameters. For example:

```
def   bridge(built, material, facility )  :

…           print 'Year Built :' , built
…           print 'Bridge Type :' , material
…           print 'Road Facility :' , facility
```

The function may be called, with the parameters in any order

```
 bridge(material = 'Steel', facility = 'I-95', built = 1989 )
```

which yields

```
Year Built : 1989
Bridge Type : Steel
Road Facility : I-95
```

If keyword arguments are used in conjunction with default arguments, some arguments may be omitted when the function is called. For example:

```
def   bridge(built, material, facility, county='Miami-Dade' )  :

…           print 'Year Built :' , built
…           print 'Bridge Type :' , material
…           print 'Road Facility :' , facility
…           print 'Maintaining Agency :' , county
```

Calling the function,

```
bridge(material = ‘Concrete’, facility = ‘I-10’, built = 2005 )
```

yields

```
Year Built : 1989
Bridge Type : Steel
Road Facility : I-95
Maintaining Agency : Miami-Dade
```

4.7 Returning a Value

The keyword *return* is used to enable a function to return a value. The *return* statement passes an expression back to the caller and then exits the function. For example:

```
def   my_address_2(housenum, streetname, city, state )  :

…         x = housenum + ‘’ + streetname + ‘, ’ + city +‘, ’ +  state
…        return x
```

Now, call the function, and pass the result (*x*) to some other variable.

```
y = my_address_2(‘1287’, ‘Napier Blvd’, ‘Gainsburg’ , ‘MN’ )
```

So, the function shall run and spit out (*return*) the result (*x*) which is assigned to the variable *y*. We can now print *y*.

```
print ‘My address is  ,’ ,  y
```

which yields

```
My address is 1287 Napier Blvd, Gainsburg, MN
```

In fact, we could have printed the returned variable *x* directly as follows.

```
print( ‘My address is  ,’ ,  my_address_2(‘1287’, ‘Napier Blvd’, ‘Gainsburg’ , ‘MN’ )  )
```

If a *return* statement has no arguments, then the function shall return *None*. If a *return* statement is not part of the function body, then the function does not return anything and it is effectively a routine. Applying a *return* statement is not required and depends on the specific needs and goals of the program.

4.8 Scope of a Variable

In the address example in the previous section, it would be tempting to print the variable *x* directly after the function has executed. For example:

```
def   my_address_2(housenum, streetname, city, state )  :

…         x = housenum + ‘’ + streetname + ‘, ’ + city +‘, ’ +  state
…        return x

# call the function to calculate x
my_address_2(‘1287’, ‘Napier Blvd’, ‘Gainsburg’ , ‘MN’ )

# print out x
print x
```

Unfortunately, this will result in the error message

```
NameError: name 'x' is not defined
```

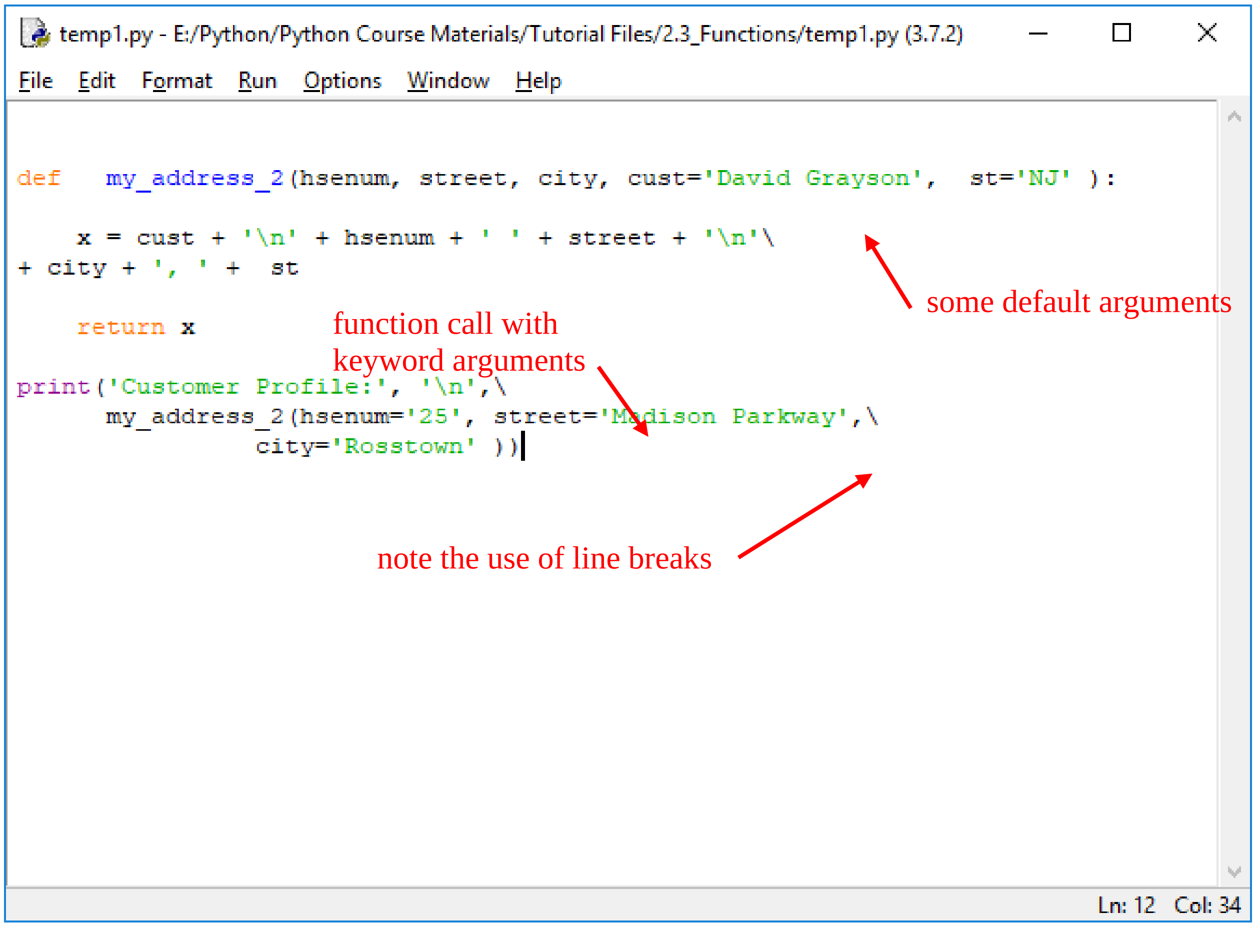
Why is this the case?
This is because variables are only accessible at certain locations within a program depending on where they were **declared** (or created). The **scope of a variable** governs where in a program a variable will be accessible and accessible to what. The scope is analogous to where the variable exists.

The *Python* interpreter recognizes **local variables** and **global variables**. A local variable is one declared within the body of a function and therefore its scope is within that function only. Outside of its **local scope** a local variable does not exist. So, once the function completes execution, the local variable "vanishes from the universe". In our example, the variable *x* was declared inside of the *my_address_2* function, therefore the interpreter cannot "recognize" it anywhere outside of the *my_address_2* function, and thus throws an error message to that effect.

A global variable on the other hand, is declared outside of a function. It has a **global scope** and therefore can be accessed inside or outside of a function.

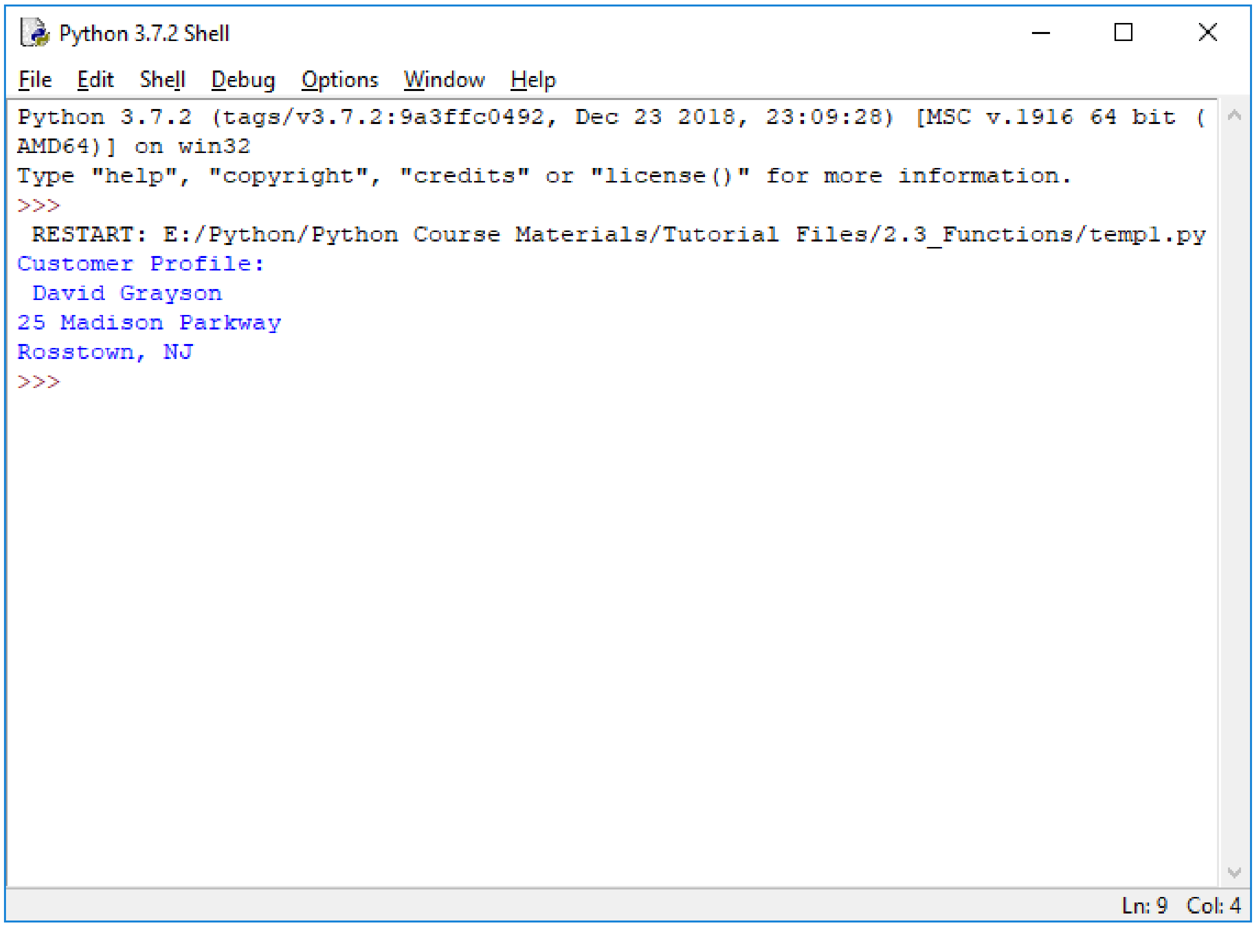
However, a variable assigned within a function will be local to that function. In order to use a global variable in a function and maintain its global character, the variable must be explicitly declared as such before being inserted into the function.

Open a new session of IDLE (Python GUI).
Open the File Editor.
In the File Editor reproduce the following code.


temp1.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/temp1.py (3.7.2)
File Edit Format Run Options Window Help
some default arguments
function call with keyword arguments
note the use of line breaks
Ln: 12 Col: 34

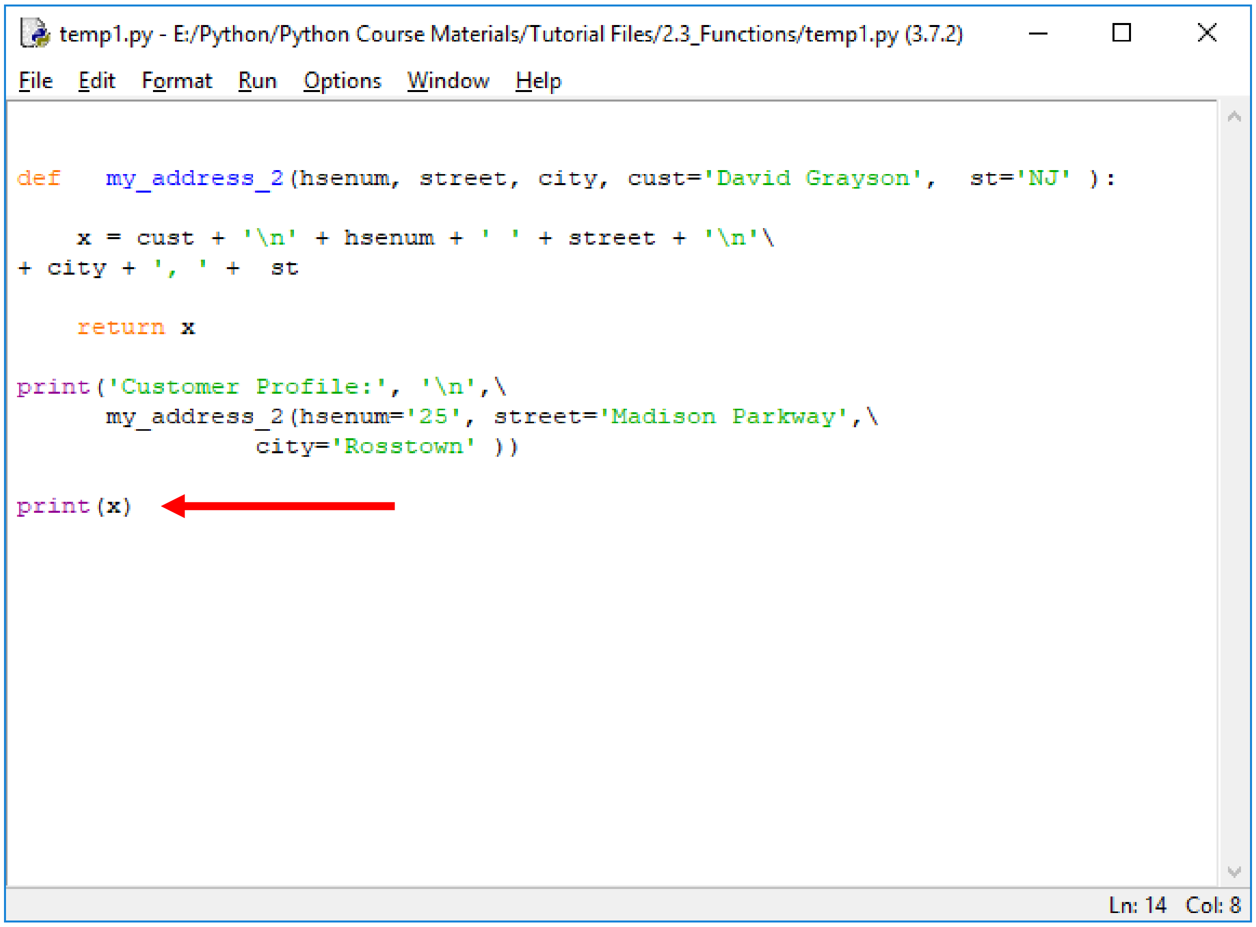
```
def   my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    x = cust + '\n' + hsenum + ' ' + street + '\n'\
+ city + ', ' +  st

    return x

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))
```

Save your file.
Run your file.
Review the results in IDLE.



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 David Grayson
25 Madison Parkway
Rosstown, NJ
>>>
```

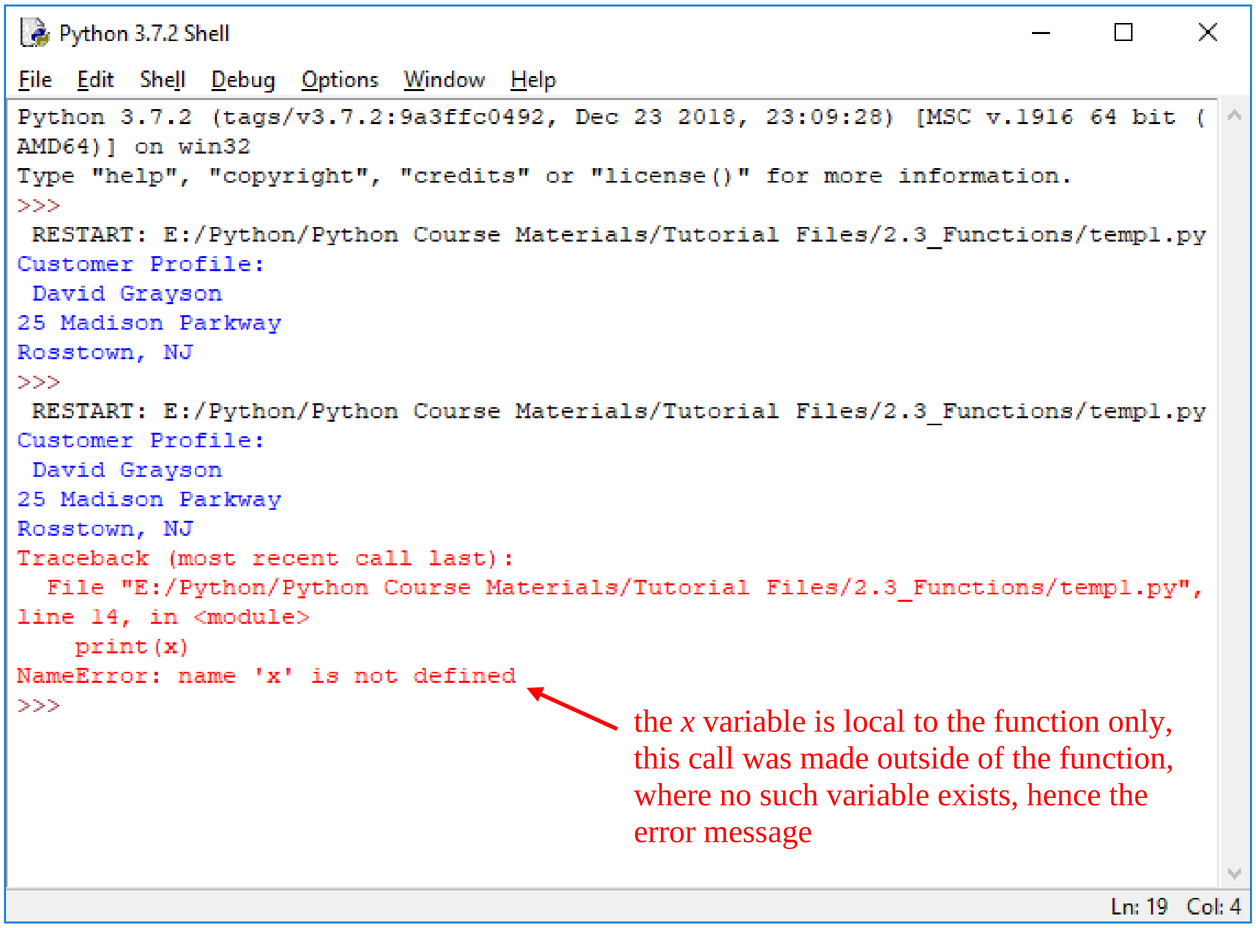
Add the following line of code.
Save your file.
Re-run your file.



temp1.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/temp1.py (3.7.2)

```
def   my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    x = cust + '\n' + hsenum + ' ' + street + '\n'\
+ city + ', ' +  st

    return x

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))

print(x)
```

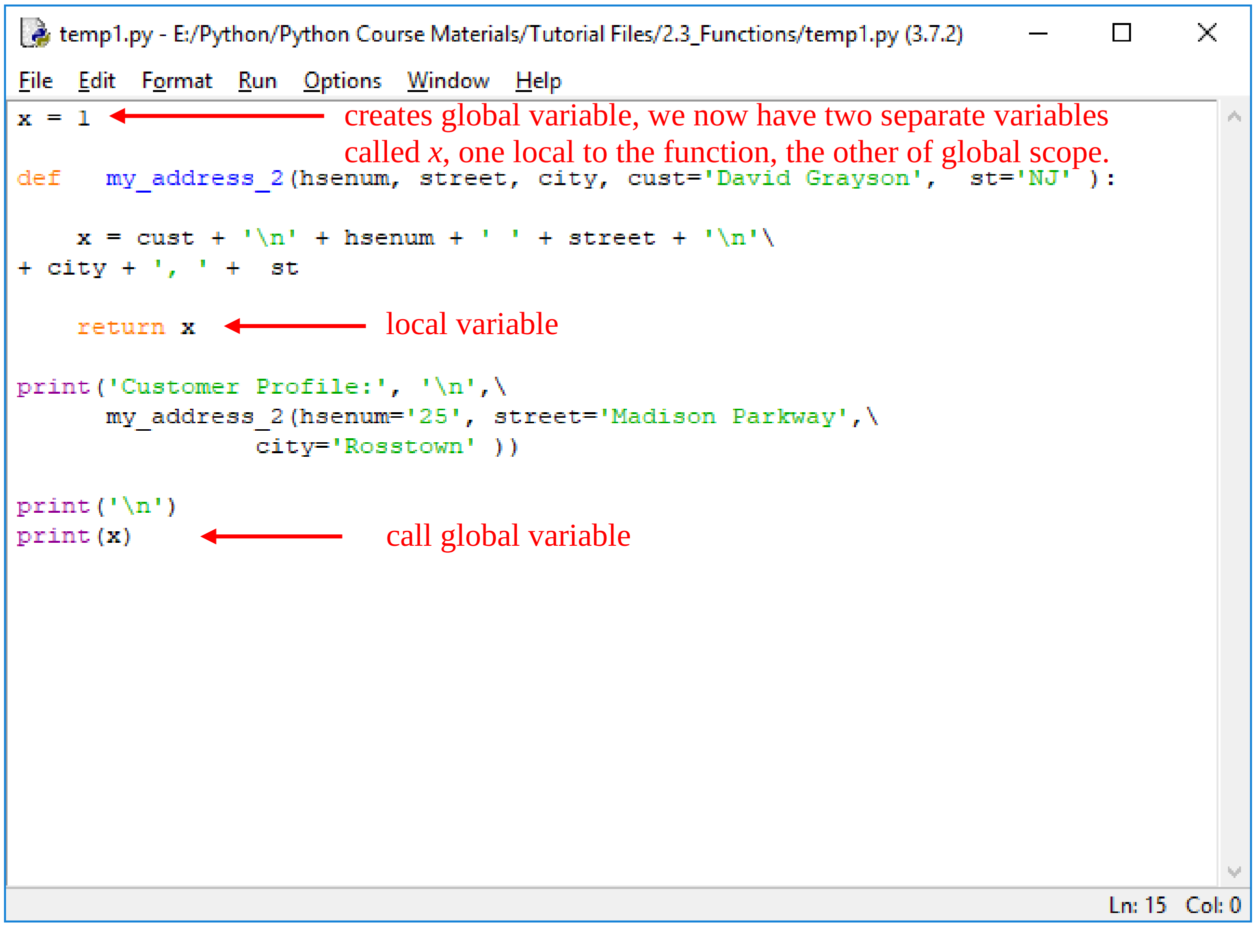
Review the results.


Python 3.7.2 Shell

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 David Grayson
25 Madison Parkway
Rosstown, NJ
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 David Grayson
25 Madison Parkway
Rosstown, NJ
Traceback (most recent call last):
  File "E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py",
line 14, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

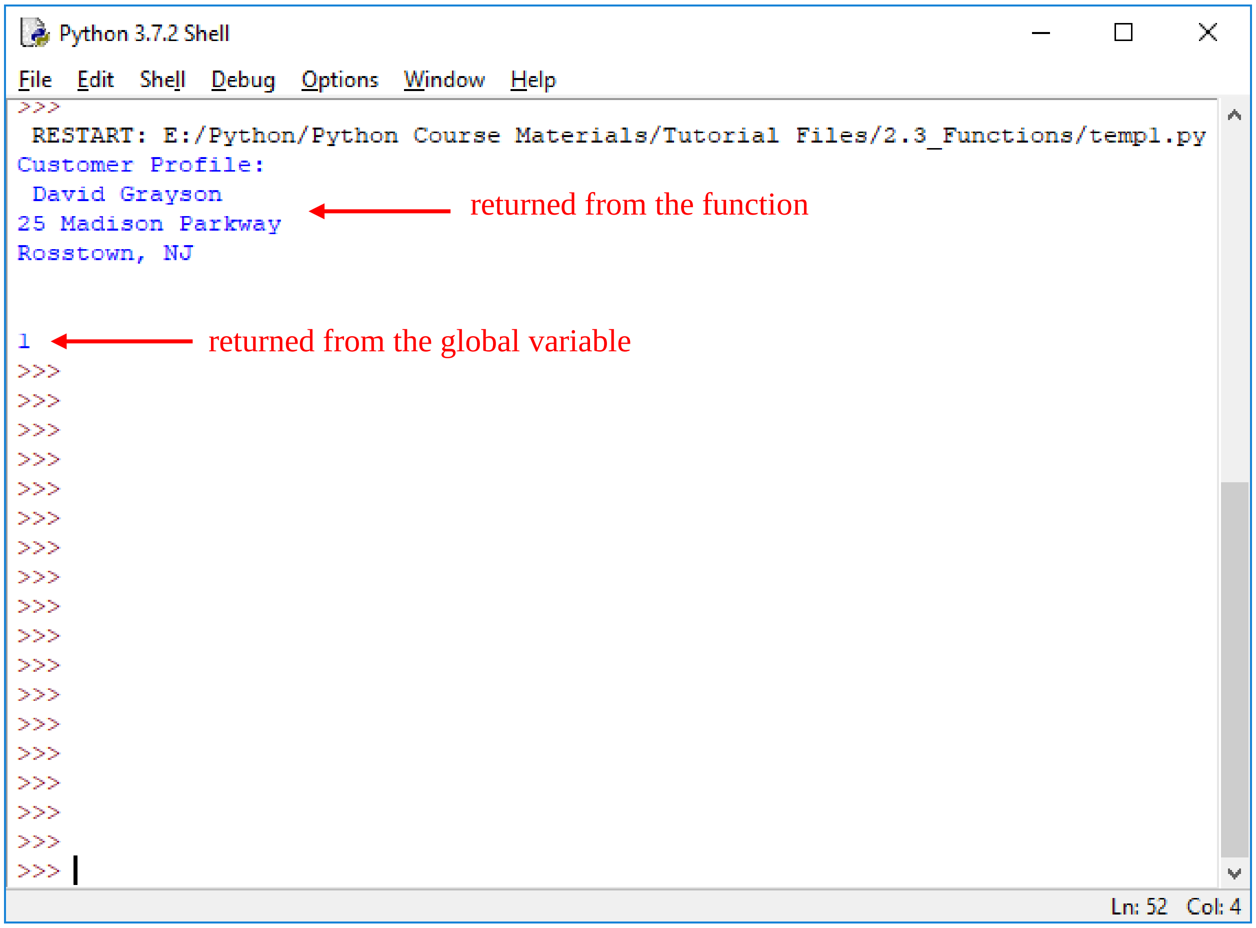
the *x* variable is local to the function only, this call was made outside of the function, where no such variable exists, hence the error message

Update the code as follows.
Save your file.
Re-run your file.


temp1.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/temp1.py (3.7.2)
File Edit Format Run Options Window Help
creates global variable, we now have two separate variables called *x*, one local to the function, the other of global scope.
local variable
call global variable
Ln: 15 Col: 0

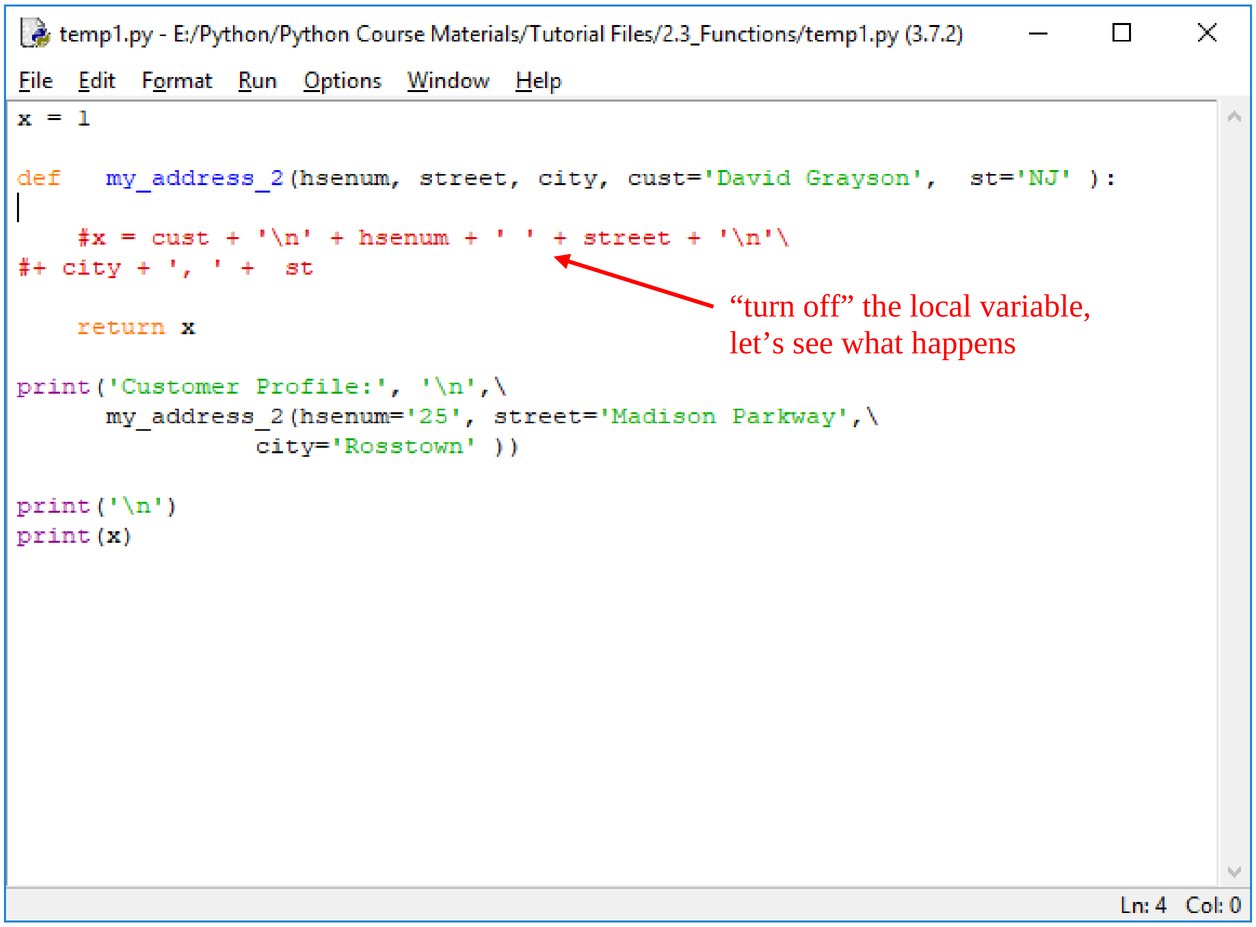
```
x = 1

def  my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    x = cust + '\n' + hsenum + ' ' + street + '\n'\
+ city + ', ' +  st

    return x

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))

print('\n')
print(x)
```

Review the results.


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
returned from the function
returned from the global variable
Ln: 52 Col: 4

```
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 David Grayson
25 Madison Parkway
Rosstown, NJ


1
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> 
```

Update the code as follows.
Save your file.
Re-run your file.

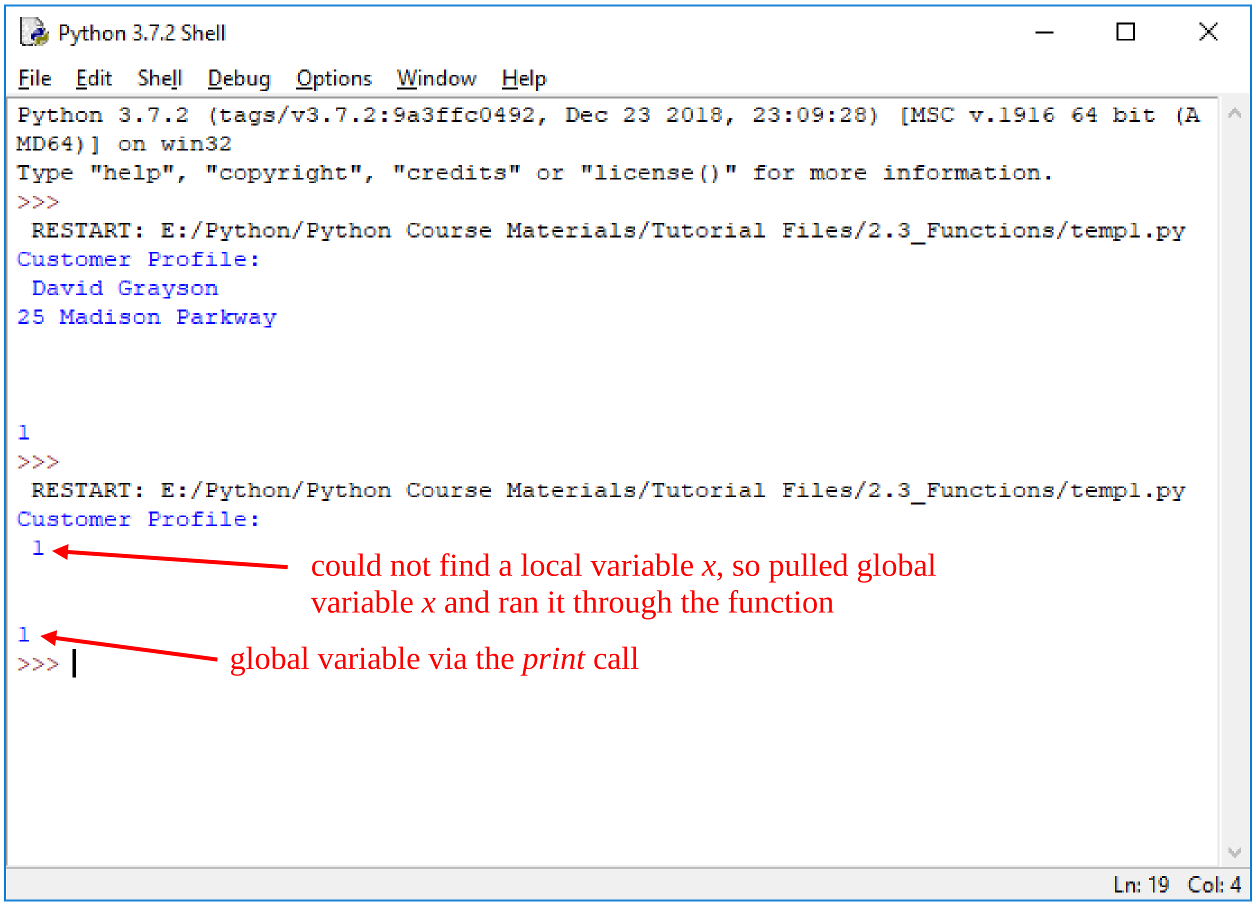


temp1.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/temp1.py (3.7.2)

```
x = 1

def   my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    #x = cust + '\n' + hsenum + ' ' + street + '\n'\
#+ city + ', ' +  st

    return x

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))

print('\n')
print(x)
```

“turn off” the local variable,
let’s see what happens

Review the results.

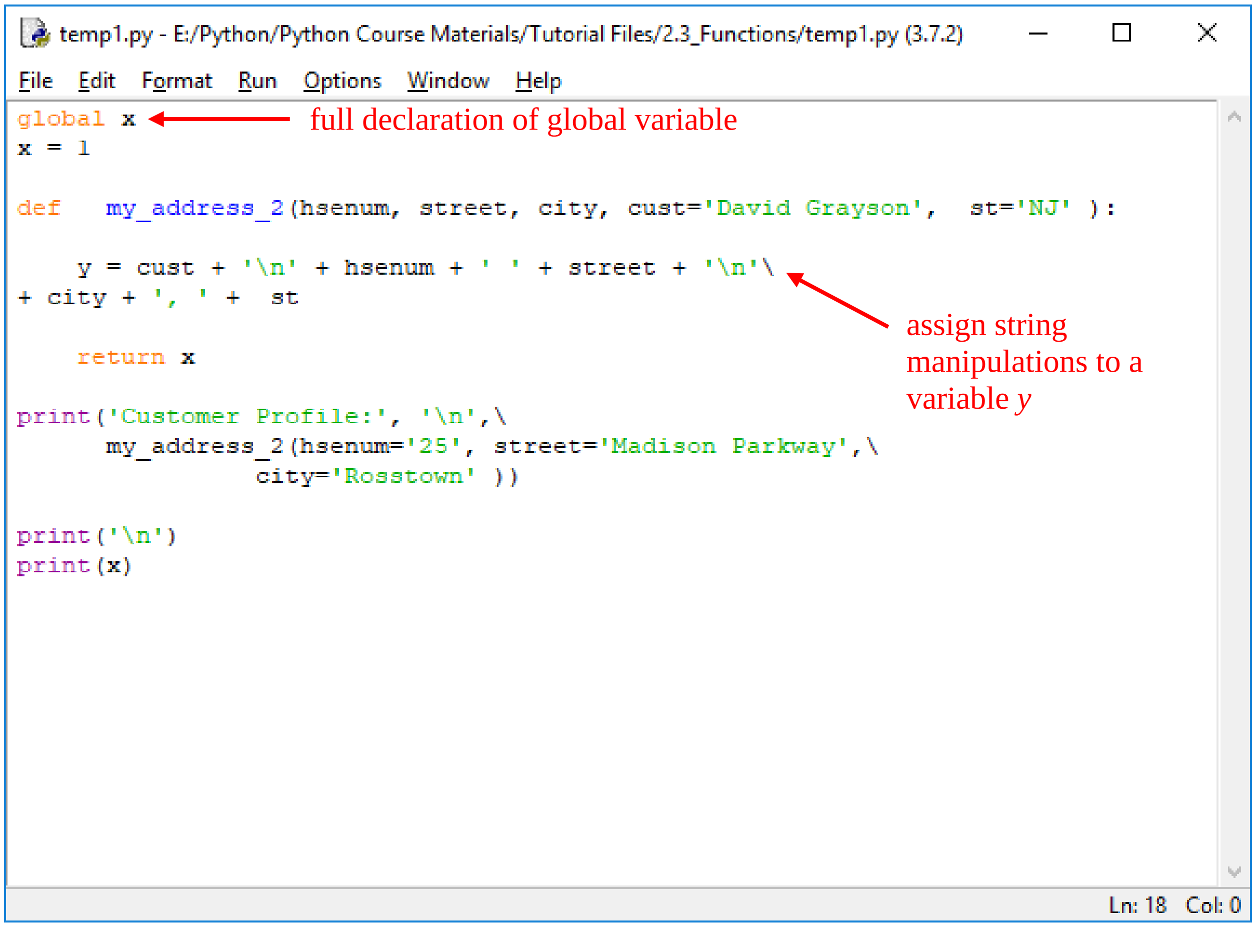


```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (A
MD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 David Grayson
25 Madison Parkway



1
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
 1

1
>>> 
```

could not find a local variable *x*, so pulled global variable *x* and ran it through the function

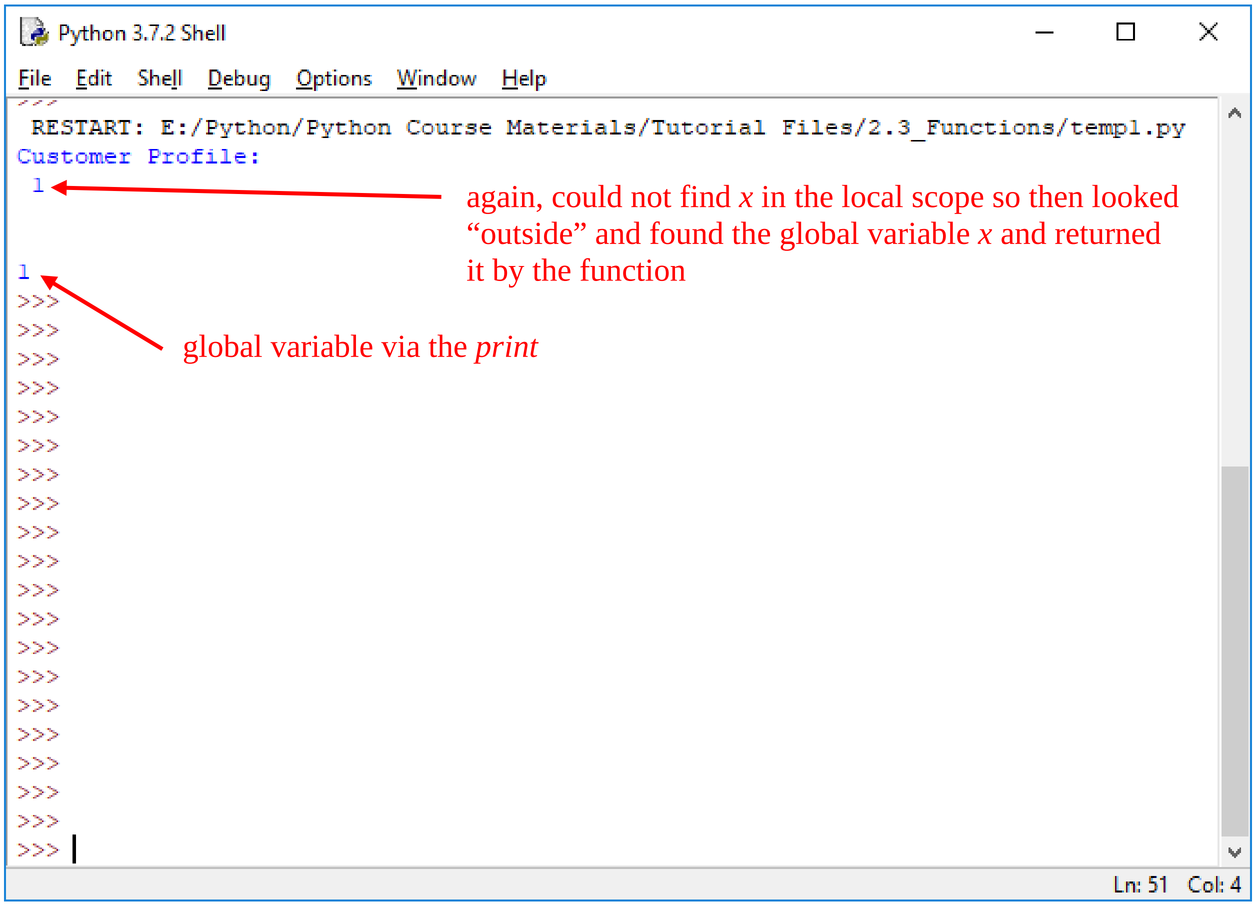
global variable via the *print* call

Update the code as follows.
Save your file.
Re-run your file.


temp1.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/temp1.py (3.7.2)
File Edit Format Run Options Window Help
full declaration of global variable
assign string manipulations to a variable *y*
Ln: 18 Col: 0

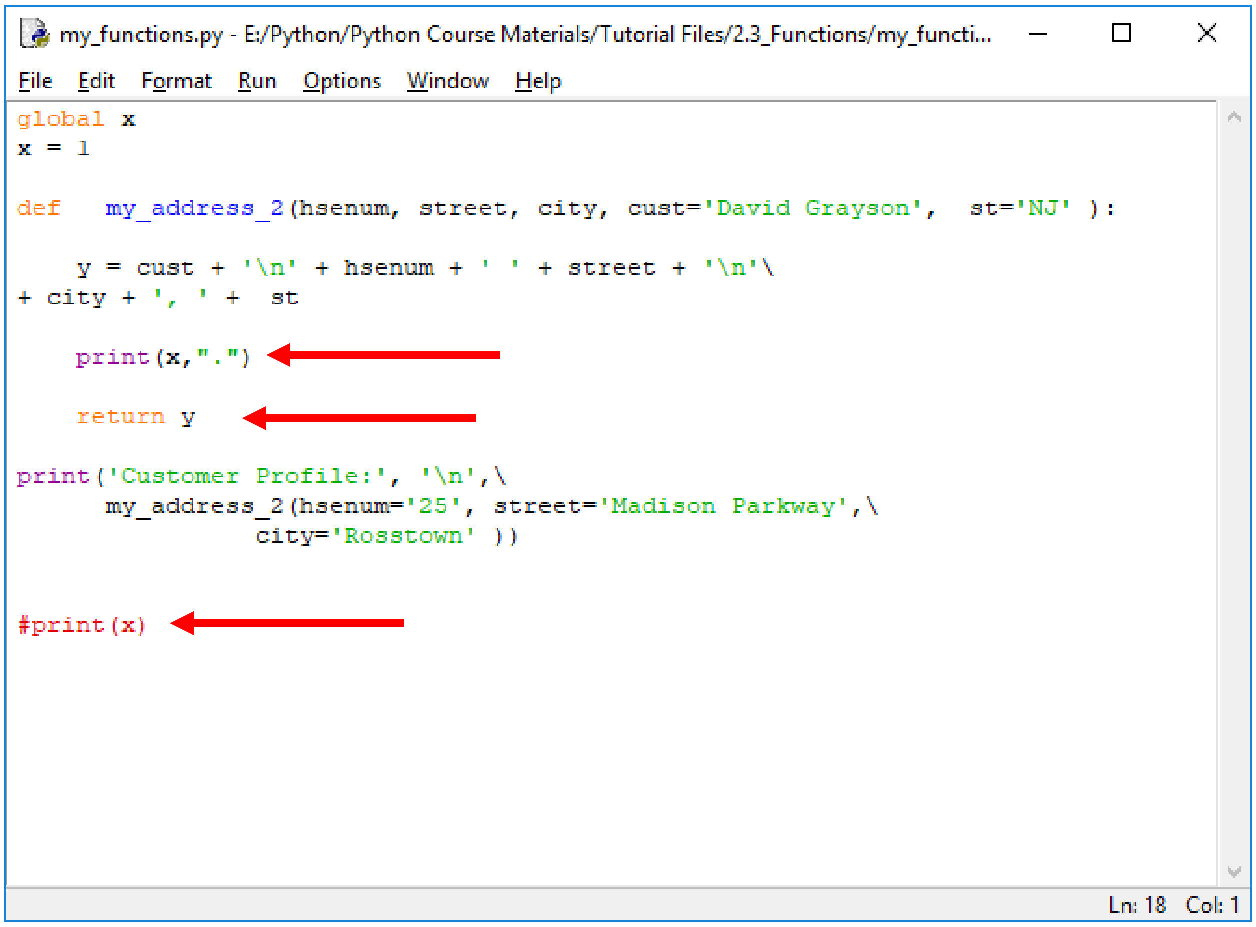
```
global x
x = 1

def   my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    y = cust + '\n' + hsenum + ' ' + street + '\n'\
+ city + ', ' +  st

    return x

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))

print('\n')
print(x)
```

Review the results.


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
again, could not find *x* in the local scope so then looked "outside" and found the global variable *x* and returned it by the function
global variable via the *print*
Ln: 51 Col: 4

```
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
Customer Profile:
  1


1
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> 
```

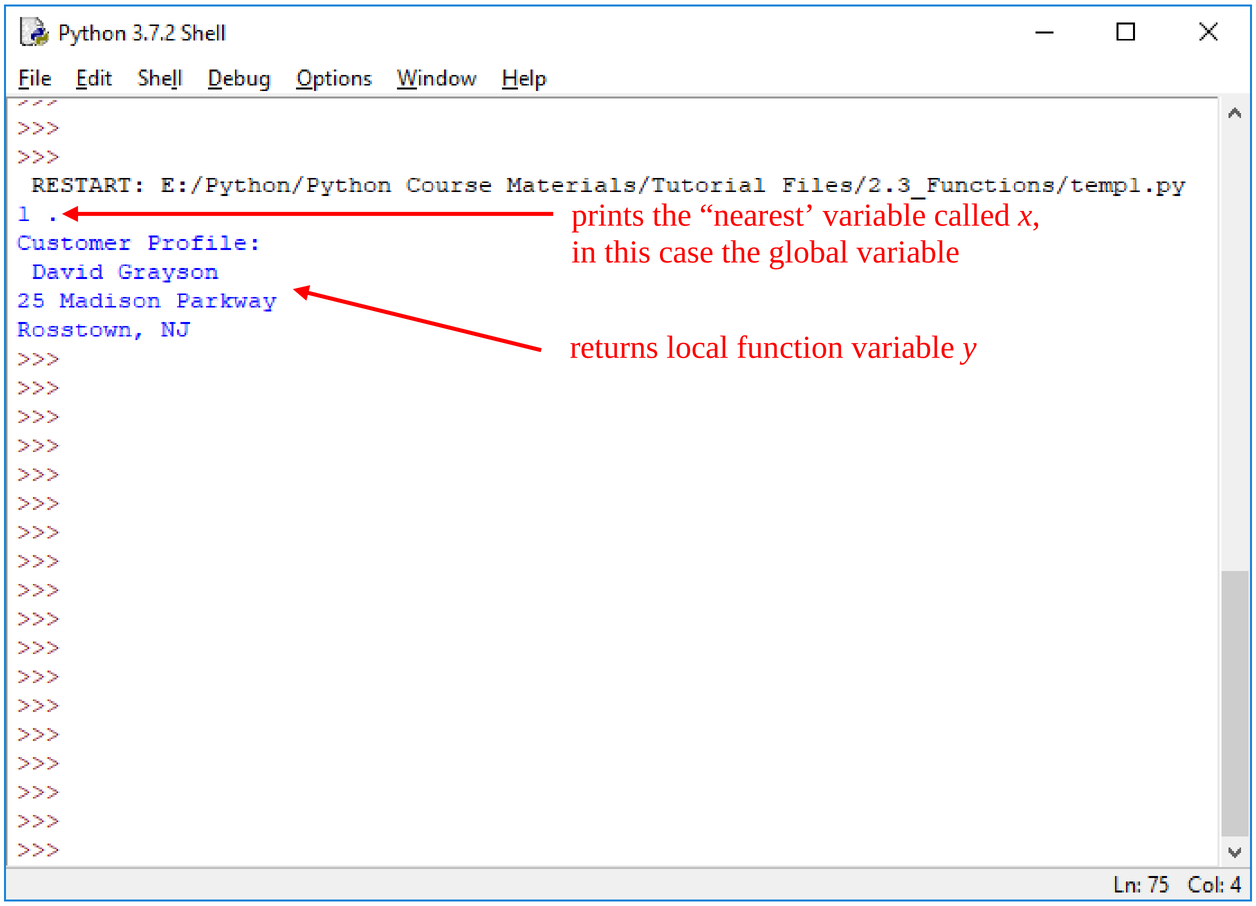
Finally, update the code as follows.
Save your file.
Re-run your file.



my_functions.py - E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/my_functi...

```
global x
x = 1

def   my_address_2(hsenum, street, city, cust='David Grayson',  st='NJ' ):

    y = cust + '\n' + hsenum + ' ' + street + '\n'\
+ city + ', ' +  st

    print(x,".")

    return y

print('Customer Profile:', '\n',\
      my_address_2(hsenum='25', street='Madison Parkway',\
                city='Rosstown' ))


#print(x)
```

Review the results.


Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
prints the "nearest' variable called *x*, in this case the global variable
returns local function variable *y*
Ln: 75 Col: 4

```
>>>
>>>
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.3_Functions/templ.py
1 .
Customer Profile:
 David Grayson
25 Madison Parkway
Rosstown, NJ
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```

4.9 Recursion

Python allows a function to be called within itself. This is called **function recursion**. Function recursion enables innovative and efficient means of looping through data to reach some result. However, function recursion is prone to the issues similar to infinite loops, and as always, the code must be meticulously checked and used with caution.

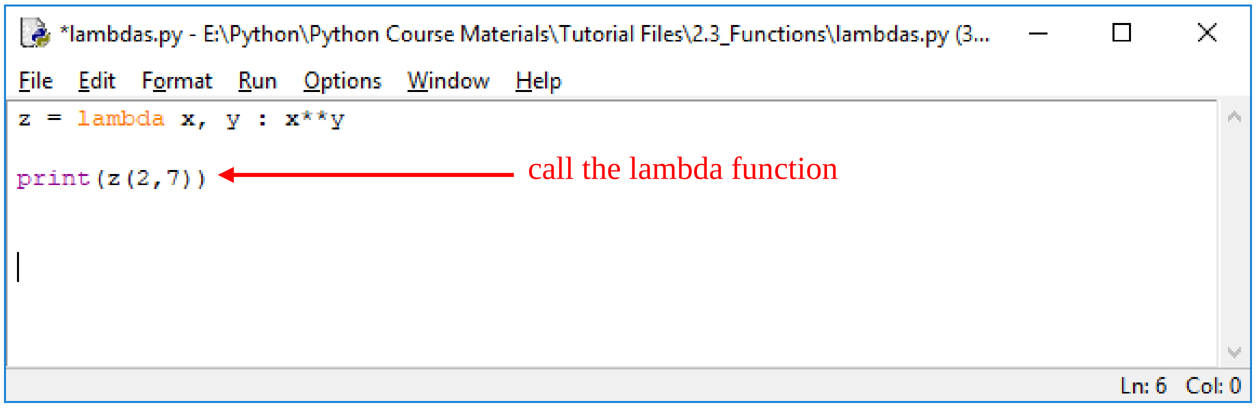
4.10 Lambda Functions

A *Python* lambda function is a small, anonymous function written over one line. They are anonymous in the sense that they do not have to be declared with the *def* statement as is required for conventional functions. A lambda function can have any number of arguments, but only one expression. The expression returns a result that is typically assigned to a variable which is used for other purposes.

The general form of the lambda function is

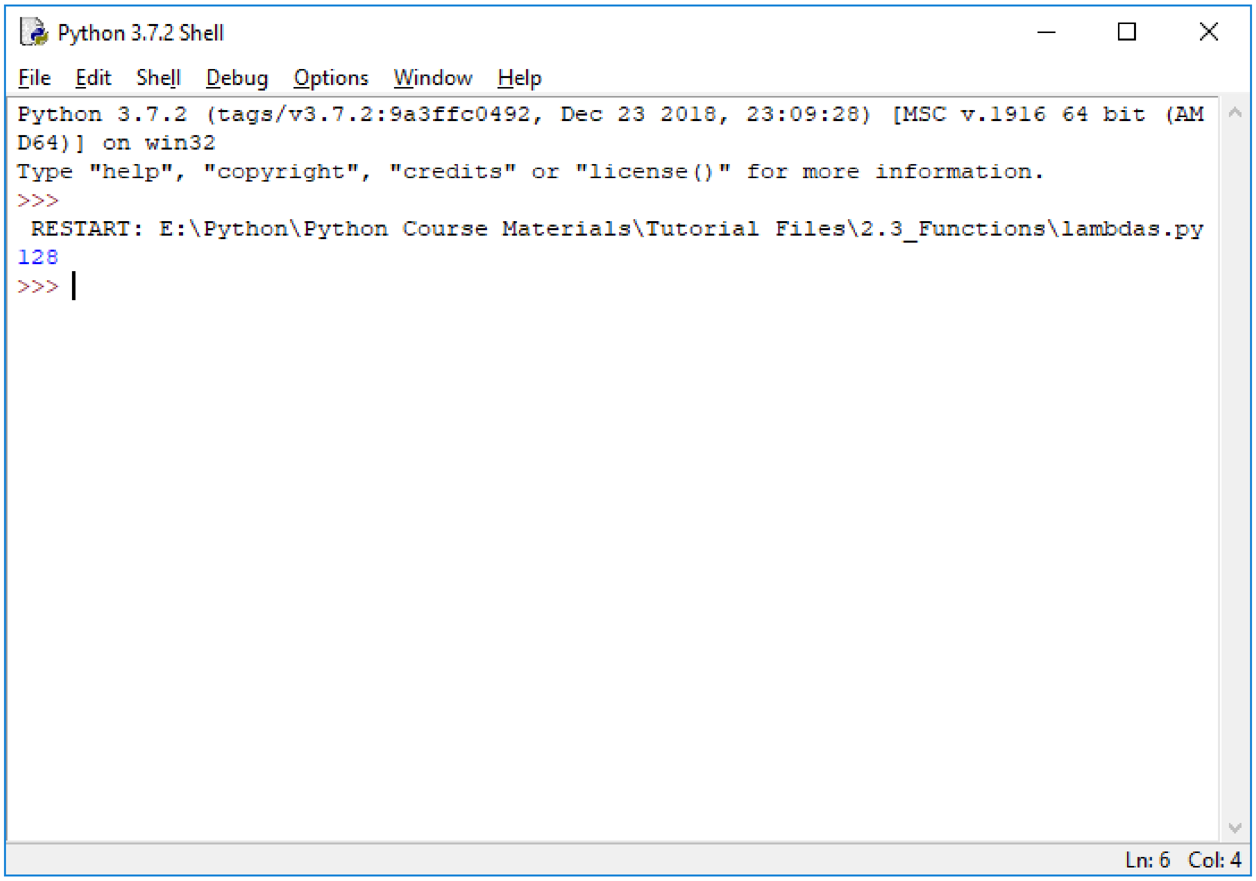
< var > = lambda < list of arguments > : < expression >

Open a new session of IDLE (Python GUI).
Open the File Editor.
In the File Editor reproduce the following code.


*lambdas.py - E:\Python\Python Course Materials\Tutorial Files\2.3_Functions\lambdas.py (3...
File Edit Format Run Options Window Help
call the lambda function
Ln: 6 Col: 0

```
z = lambda x, y : x**y

print(z(2,7))
```

This yields the following result.



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.3_Functions\lambdas.py
128
>>> 
```

5. INPUT AND OUTPUT (I/O) FUNCTIONS

5.1 Introduction

The built-in input and output (I/O) functions enable an executing program to communicate with the user. Input functions enable the user to supply data to the program, whereas output functions enable the program to display results back to the user.

Input may be from the user entering data via their keyboard, but it may also be from a file, a database or some other external source. Output may be displayed to the console, or to the screen via a graphical user interface (GUI), or it may be sent to a file, a database or some other external repository.

5.2 The *input( )* Function

The simplest way to send data to an executing program is through the *input( )* function. The *input( )* function reads one line of data from the keyboard. *The input( )* function causes the program execution to pause while the user types in the data via the keyboard and hits the **Enter** key on the keyboard which then causes the program to resume execution.

The *input( )* function reads and returns all characters typed in as a string.

The general application of the *input( )* is as follows.

< var > = input(< prompt >)

where < *prompt* > is an optional prompt, and the data entered via the keyboard is assigned to the variable < *var* > as a string.

5.3 Casting

The following built-in functions can be used to convert the string returned by the *input( )* function to the appropriate data type.

Constructor	Description
int( )	constructs an integer type from a float literal (by rounding down to the next whole number) or a string literal representation of a whole number
float( )	constructs a floating number type from an integer or float literal or a string literal representation of a floating number or integer
str( )	constructs a string type from another data type
complex( )	constructs a complex number from a string literal representation of complex number, or it returns a complex number when the real and imaginary parts are supplied

Specifying the data type is called **casting**.

5.4 The *print( )* Function

The *print( )* function displays or presents data back to the user.

The syntax is of the form

print (< object1 > , < object2 > , …)

An < *object* > may be of any data type. The *print( )* function returns a string representation of the string concatenation of the objects. The former syntax is a legacy from previous builds of *Python*. The latter syntax is more recent and more versatile as it allows for the incorporation of **keyword arguments** that control the format of the output.

It is pertinent to note that when reviewing the *Python* literature, it is common to see the print command used in the form

print < object1 >, < object2 >, …

This was the standard in previous versions of *Python* up to *Python 2*. As of *Python 3*, the use of the parentheses is incorporated.

The syntax for the *print( )* with keyword arguments is of the form

print (< object > , … , < object >, < keyword > = < value >)

Examples of keywords in the print function include:

Keyword Argument	Description	Example
sep=	causes each object in the output string to be separated by a specified string instead of the default single space	*print(<obj>, … <obj>, sep=<str>*
end=	causes output of multiple *print* calls to be displayed on one line and separated by <*str*> rather than the default separate line for each *print* call	*print(<obj>, … <obj>, end=<str>*

Example:

Set up a simple calculator that enables a Fire Protection Engineer to compute the portion of the needed fire flow (NFF) (in gallons per minute) attributed to the area and the construction classification of the building.

According the Insurance Services Office (ISO), the portion of the NFF attributed to the area and the construction classification of the building (*C*), in gallons per minute, is given by the empirical formula

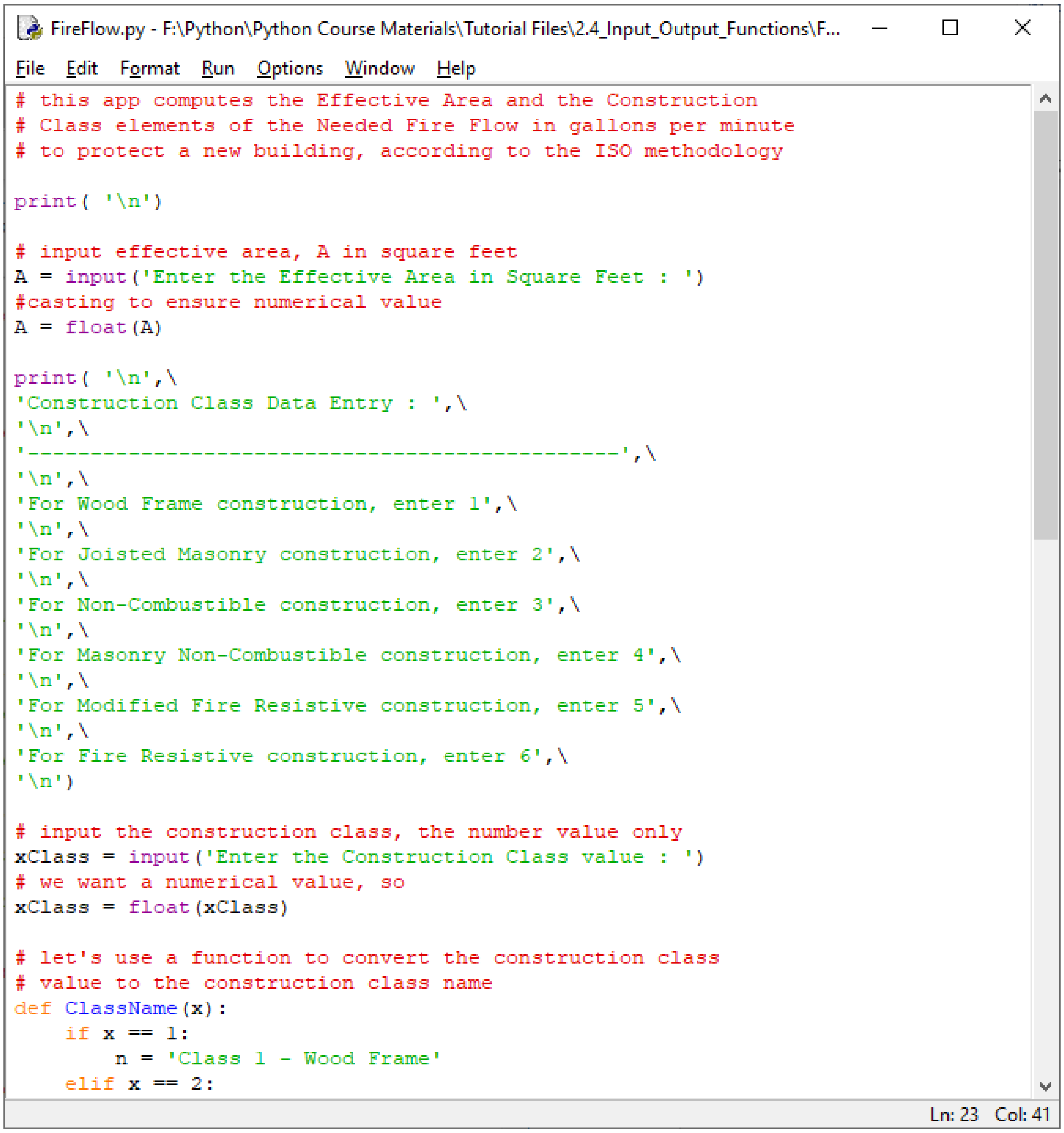
$$C = 18F\sqrt{A}$$

where

A = effective area in square feet
F = the construction coefficient
= 1.5 for Construction Class 1 – Wood Frame
= 1.0 for Construction Class 2 – Joisted masonry
= 0.8 for Construction Class 3 – Non-Combustible
= 0.8 for Construction Class 4 – Masonry Non-Combustible
= 0.6 for Construction Class 5 – Modified Fire Resistive
= 0.6 for Construction Class 6 – Fire Resistive

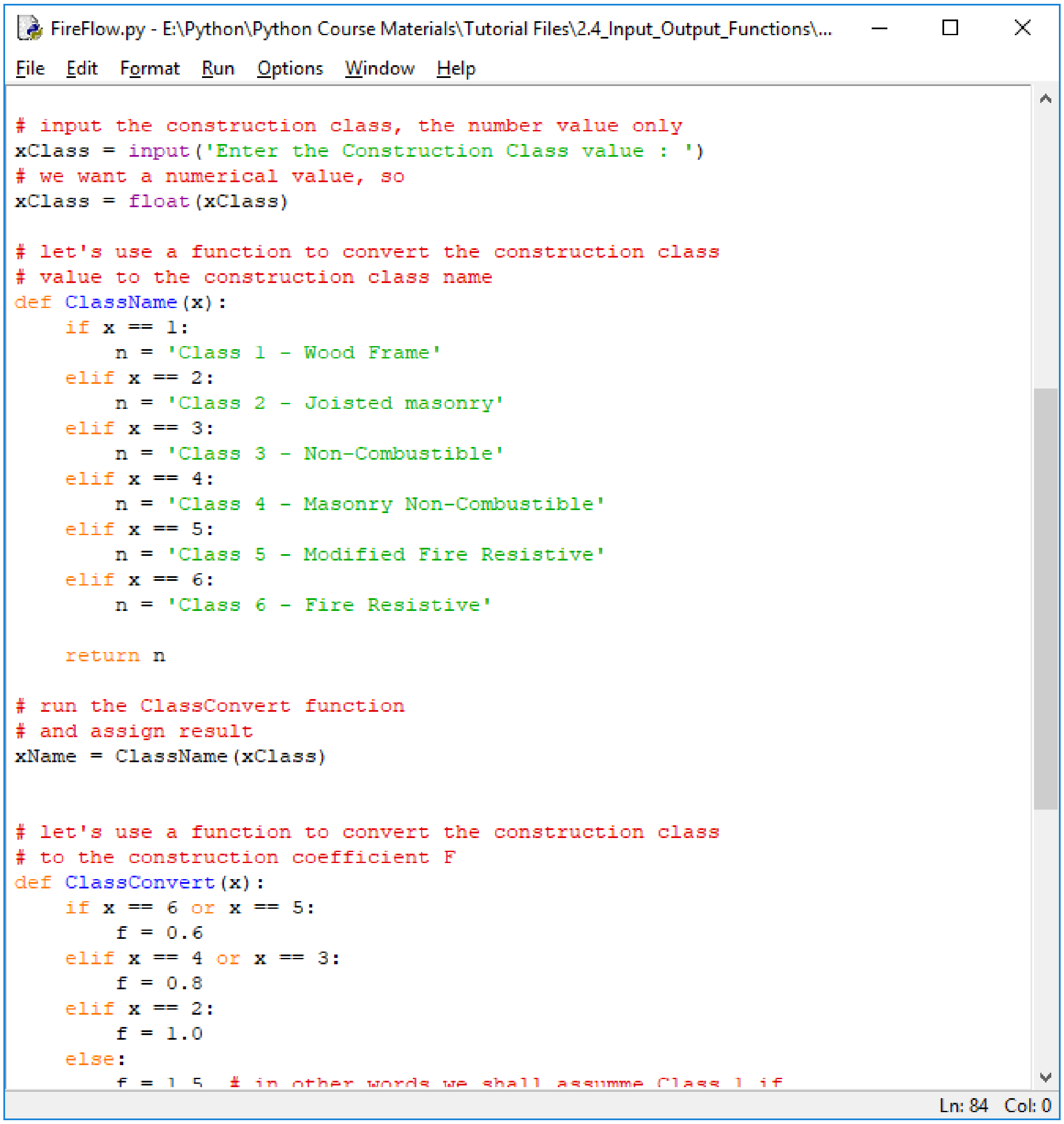
(Note: In this demonstration we shall keep it simple. However, please add your own innovations and be creative.)

Open a new session of IDLE (Python GUI).
Open the File Editor.
In the File Editor reproduce the following code.


FireFlow.py - F:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Functions\F...
File Edit Format Run Options Window Help
Ln: 23 Col: 41

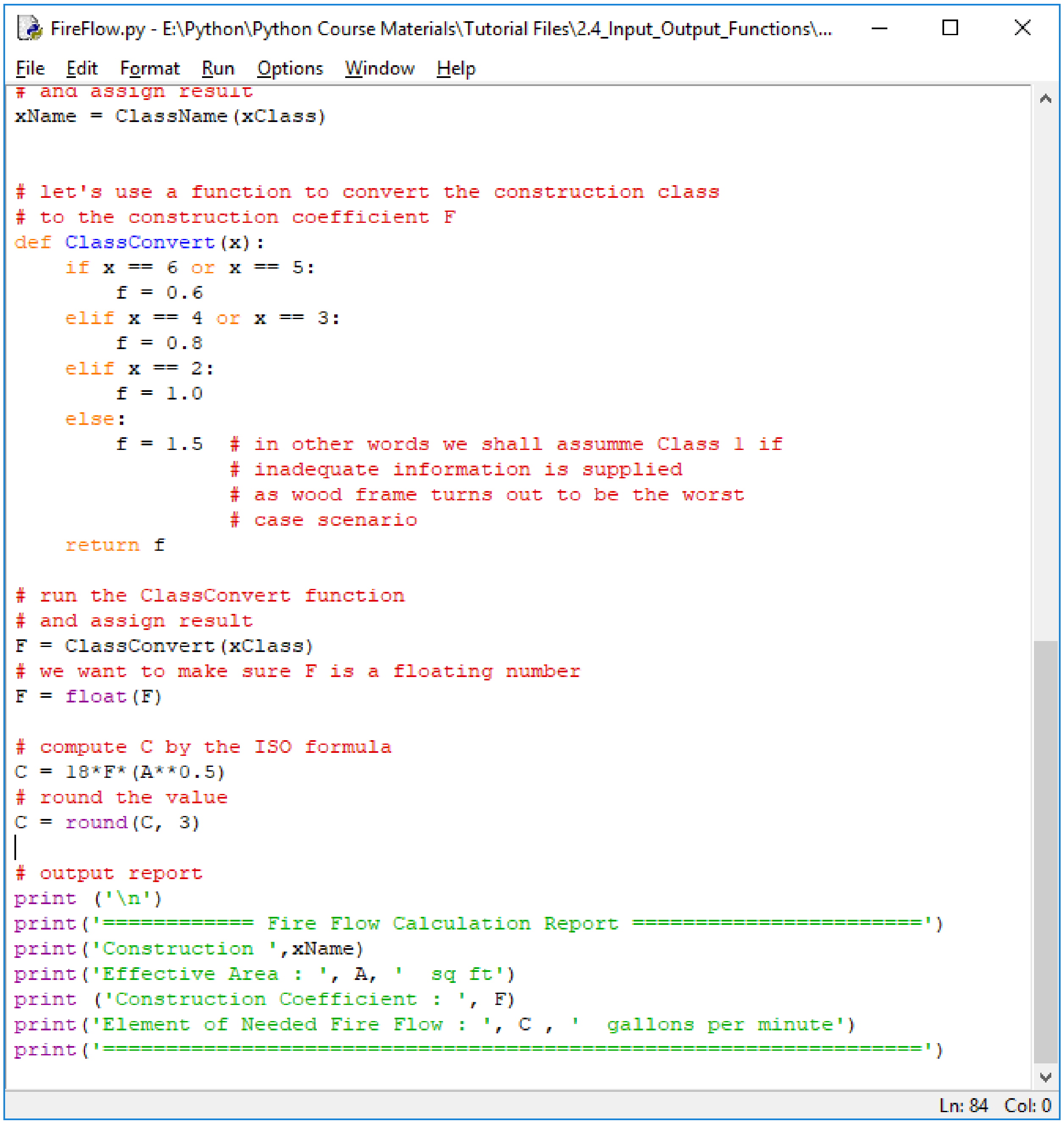
```python
# this app computes the Effective Area and the Construction
# Class elements of the Needed Fire Flow in gallons per minute
# to protect a new building, according to the ISO methodology

print( '\n')

# input effective area, A in square feet
A = input('Enter the Effective Area in Square Feet : ')
#casting to ensure numerical value
A = float(A)

print( '\n',\
'Construction Class Data Entry : ',\
'\n',\
'-----------------------------------------------',\
'\n',\
'For Wood Frame construction, enter 1',\
'\n',\
'For Joisted Masonry construction, enter 2',\
'\n',\
'For Non-Combustible construction, enter 3',\
'\n',\
'For Masonry Non-Combustible construction, enter 4',\
'\n',\
'For Modified Fire Resistive construction, enter 5',\
'\n',\
'For Fire Resistive construction, enter 6',\
'\n')

# input the construction class, the number value only
xClass = input('Enter the Construction Class value : ')
# we want a numerical value, so
xClass = float(xClass)

# let's use a function to convert the construction class
# value to the construction class name
def ClassName(x):
    if x == 1:
        n = 'Class 1 - Wood Frame'
    elif x == 2:
```

Continue as follows.


FireFlow.py - E:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Functions\...
File Edit Format Run Options Window Help
Ln: 84 Col: 0

```python
# input the construction class, the number value only
xClass = input('Enter the Construction Class value : ')
# we want a numerical value, so
xClass = float(xClass)

# let's use a function to convert the construction class
# value to the construction class name
def ClassName(x):
    if x == 1:
        n = 'Class 1 - Wood Frame'
    elif x == 2:
        n = 'Class 2 - Joisted masonry'
    elif x == 3:
        n = 'Class 3 - Non-Combustible'
    elif x == 4:
        n = 'Class 4 - Masonry Non-Combustible'
    elif x == 5:
        n = 'Class 5 - Modified Fire Resistive'
    elif x == 6:
        n = 'Class 6 - Fire Resistive'

    return n

# run the ClassConvert function
# and assign result
xName = ClassName(xClass)


# let's use a function to convert the construction class
# to the construction coefficient F
def ClassConvert(x):
    if x == 6 or x == 5:
        f = 0.6
    elif x == 4 or x == 3:
        f = 0.8
    elif x == 2:
        f = 1.0
    else:
        f = 1.5  # in other words we shall assumme Class 1 if
```

Continue as follows.

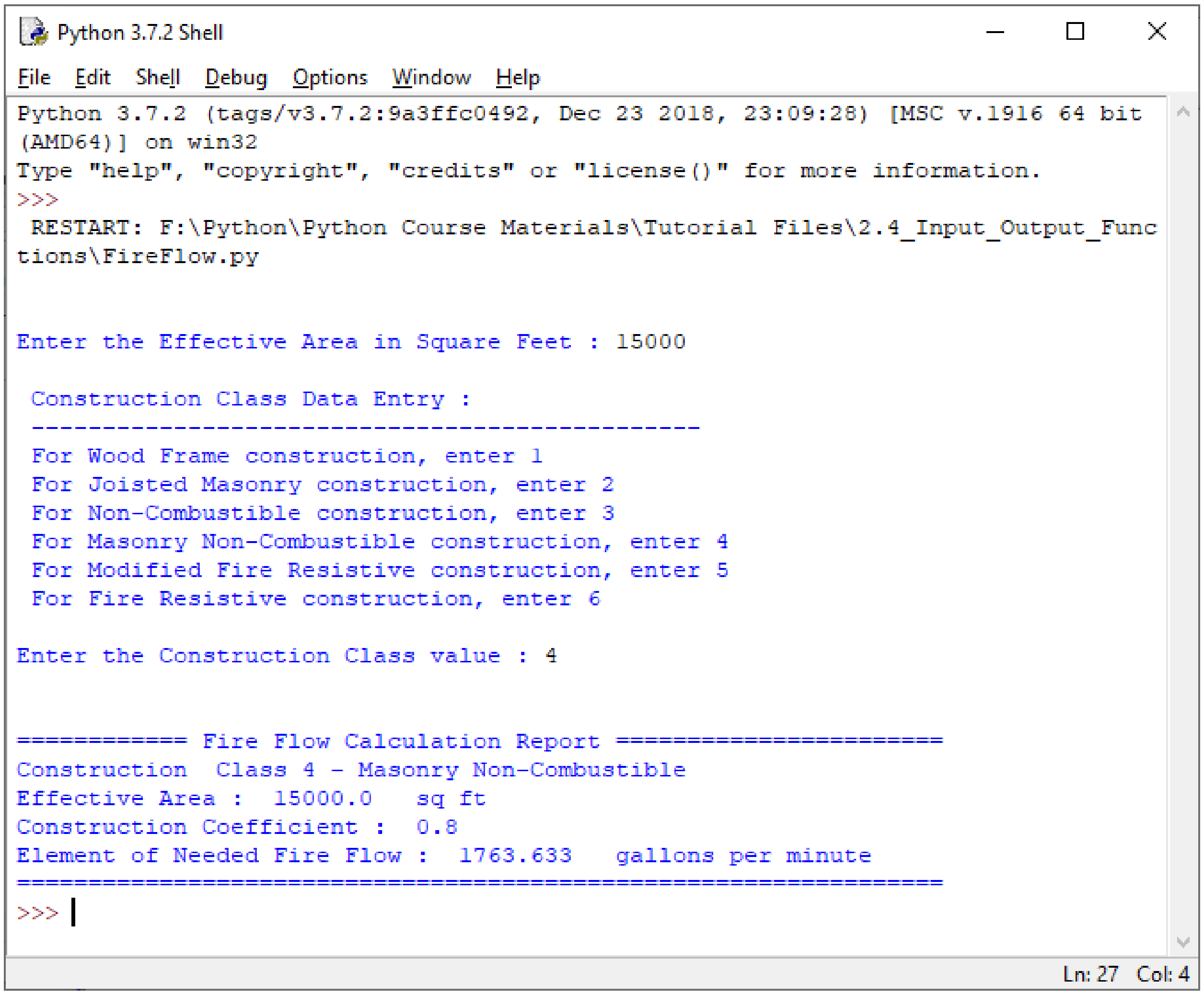

FireFlow.py - E:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Functions\...

File Edit Format Run Options Window Help

```
# and assign result
xName = ClassName(xClass)


# let's use a function to convert the construction class
# to the construction coefficient F
def ClassConvert(x):
    if x == 6 or x == 5:
        f = 0.6
    elif x == 4 or x == 3:
        f = 0.8
    elif x == 2:
        f = 1.0
    else:
        f = 1.5  # in other words we shall assumme Class 1 if
                 # inadequate information is supplied
                 # as wood frame turns out to be the worst
                 # case scenario
    return f

# run the ClassConvert function
# and assign result
F = ClassConvert(xClass)
# we want to make sure F is a floating number
F = float(F)

# compute C by the ISO formula
C = 18*F*(A**0.5)
# round the value
C = round(C, 3)

# output report
print ('\n')
print('============ Fire Flow Calculation Report =======================')
print('Construction ',xName)
print('Effective Area : ', A, '  sq ft')
print ('Construction Coefficient : ', F)
print('Element of Needed Fire Flow : ', C , '  gallons per minute')
print('==================================================================')
```

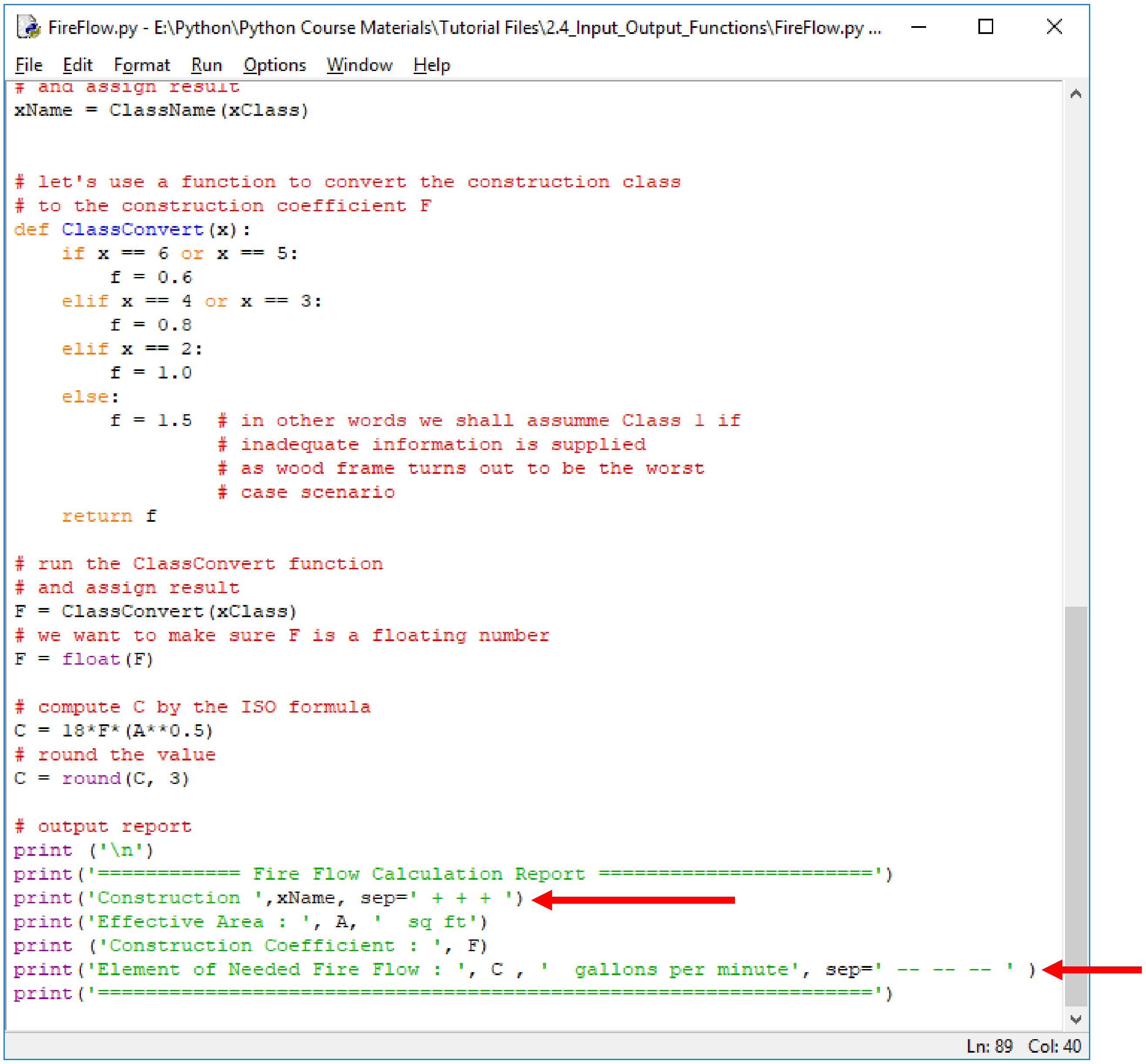
Ln: 84 Col: 0

Run the program.
Pick some random entry values at the prompts to test run the program to completion.



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: F:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Func
tions\FireFlow.py


Enter the Effective Area in Square Feet : 15000

 Construction Class Data Entry :
 -----------------------------------------------
 For Wood Frame construction, enter 1
 For Joisted Masonry construction, enter 2
 For Non-Combustible construction, enter 3
 For Masonry Non-Combustible construction, enter 4
 For Modified Fire Resistive construction, enter 5
 For Fire Resistive construction, enter 6

Enter the Construction Class value : 4


============ Fire Flow Calculation Report =======================
Construction  Class 4 - Masonry Non-Combustible
Effective Area :  15000.0   sq ft
Construction Coefficient :  0.8
Element of Needed Fire Flow :  1763.633   gallons per minute
=================================================================
>>> |
```

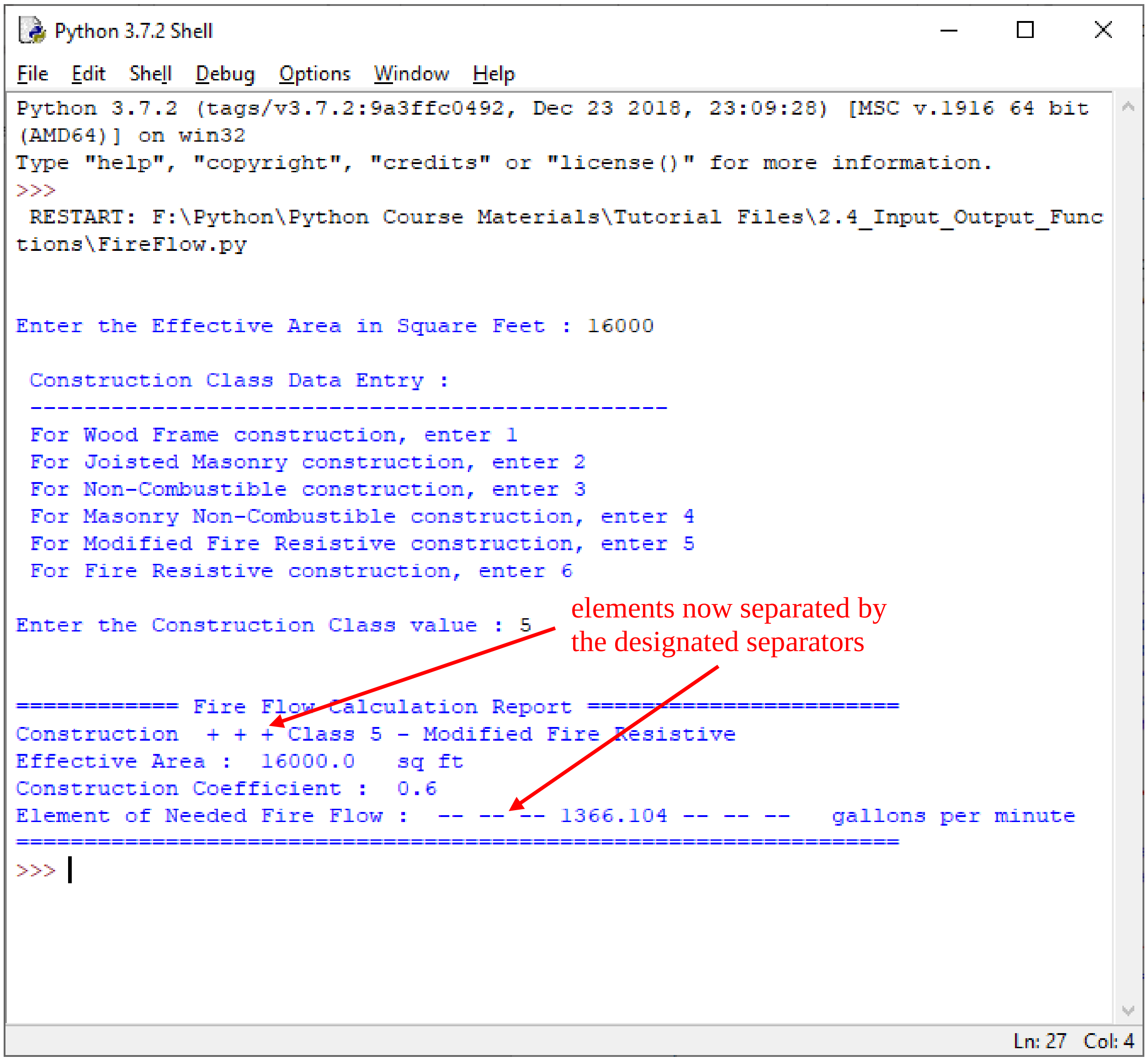
Implement the following updates.



FireFlow.py - E:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Functions\FireFlow.py ...
File Edit Format Run Options Window Help
Ln: 89 Col: 40

```
# and assign result
xName = ClassName(xClass)


# let's use a function to convert the construction class
# to the construction coefficient F
def ClassConvert(x):
    if x == 6 or x == 5:
        f = 0.6
    elif x == 4 or x == 3:
        f = 0.8
    elif x == 2:
        f = 1.0
    else:
        f = 1.5  # in other words we shall assumme Class 1 if
                 # inadequate information is supplied
                 # as wood frame turns out to be the worst
                 # case scenario
    return f

# run the ClassConvert function
# and assign result
F = ClassConvert(xClass)
# we want to make sure F is a floating number
F = float(F)

# compute C by the ISO formula
C = 18*F*(A**0.5)
# round the value
C = round(C, 3)

# output report
print ('\n')
print('============ Fire Flow Calculation Report =======================')
print('Construction ',xName, sep=' + + + ')
print('Effective Area : ', A, '  sq ft')
print ('Construction Coefficient : ', F)
print('Element of Needed Fire Flow : ', C , '  gallons per minute', sep=' -- -- -- ' )
print('=================================================================')
```

Run the program.
Review the changes to the output report.



```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
 RESTART: F:\Python\Python Course Materials\Tutorial Files\2.4_Input_Output_Func
tions\FireFlow.py


Enter the Effective Area in Square Feet : 16000

 Construction Class Data Entry :
 -----------------------------------------------
 For Wood Frame construction, enter 1
 For Joisted Masonry construction, enter 2
 For Non-Combustible construction, enter 3
 For Masonry Non-Combustible construction, enter 4
 For Modified Fire Resistive construction, enter 5
 For Fire Resistive construction, enter 6

Enter the Construction Class value : 5


============ Fire Flow Calculation Report =======================
Construction  + + + Class 5 - Modified Fire Resistive
Effective Area :  16000.0   sq ft
Construction Coefficient :  0.6
Element of Needed Fire Flow :  -- -- -- 1366.104 -- -- --   gallons per minute
=================================================================
>>> 
```

On your own, think of other ways to modify the output report to a preferred style. How about using an f-string, or the *string.format( )* method?

6. MODULES

6.1 Introduction

A module is a Python file (suffix **.py**) that contains functions that a user can call and use in another application. A module is therefore essentially a code library.

6.2 Calling a Module

A function in a module is called by an *import* statement followed by a call to the function, with the module name prefixed to the function name. The function call must supply any relevant arguments to the function. The syntax is of the set up

import < filename >

< filename > . < functioname >(arguments)

Typically, the result returned by the function will be assigned to some variable in the current application. Therefore, the common set up will be of the form

import < filename >

< var > = < filename > . < functioname >(arguments)

6.3 Naming and Renaming a Module

Any admissible file name with the Python **.py** suffix can be used as module file. A module file may be given an alias at the time of import using the *as* keyword. For example:

import < filename > as alias

< var > = < alias > . < functioname >(arguments)

6.4 Variables in Modules

In addition to functions, variables of all types – lists, tuples, dictionaries, etc., can be called and used via a module file.

6.5 Built-in in Modules

Python has several built-in modules. The following is a very limited selection of built-in modules. Please consult the Python literature or a simple google search to review the extensive Python built-in modules.

Module	Description
calender	functions for working with calendars and dates
cmath	mathematical functions for working with complex numbers
email	package for parsing, manipulating, and generating email messages.
fraction	functions for working with rational numbers
html	package for and web pages
linecache	provides random access to individual lines of code from text files
math	mathematical functions such as in trigonometry, geometry etc., etc.
os	for accessing operating system interfaces
random	for generating random numbers from various distributions
statistics	statistical functions
string	string functions
tkinter	package for developing graphical user interfaces

6.6 The *dir( )* Function

The *dir( )* function is a built-in function that returns a list of all function names and variables in a module. The *dir( )* function may be used on user-made modules as well as built-in modules.

The general form of the use of the *dir( )* function is as follows.

```
import  < filename >

 < var > = dir(< filename >)

print(< var >)
```

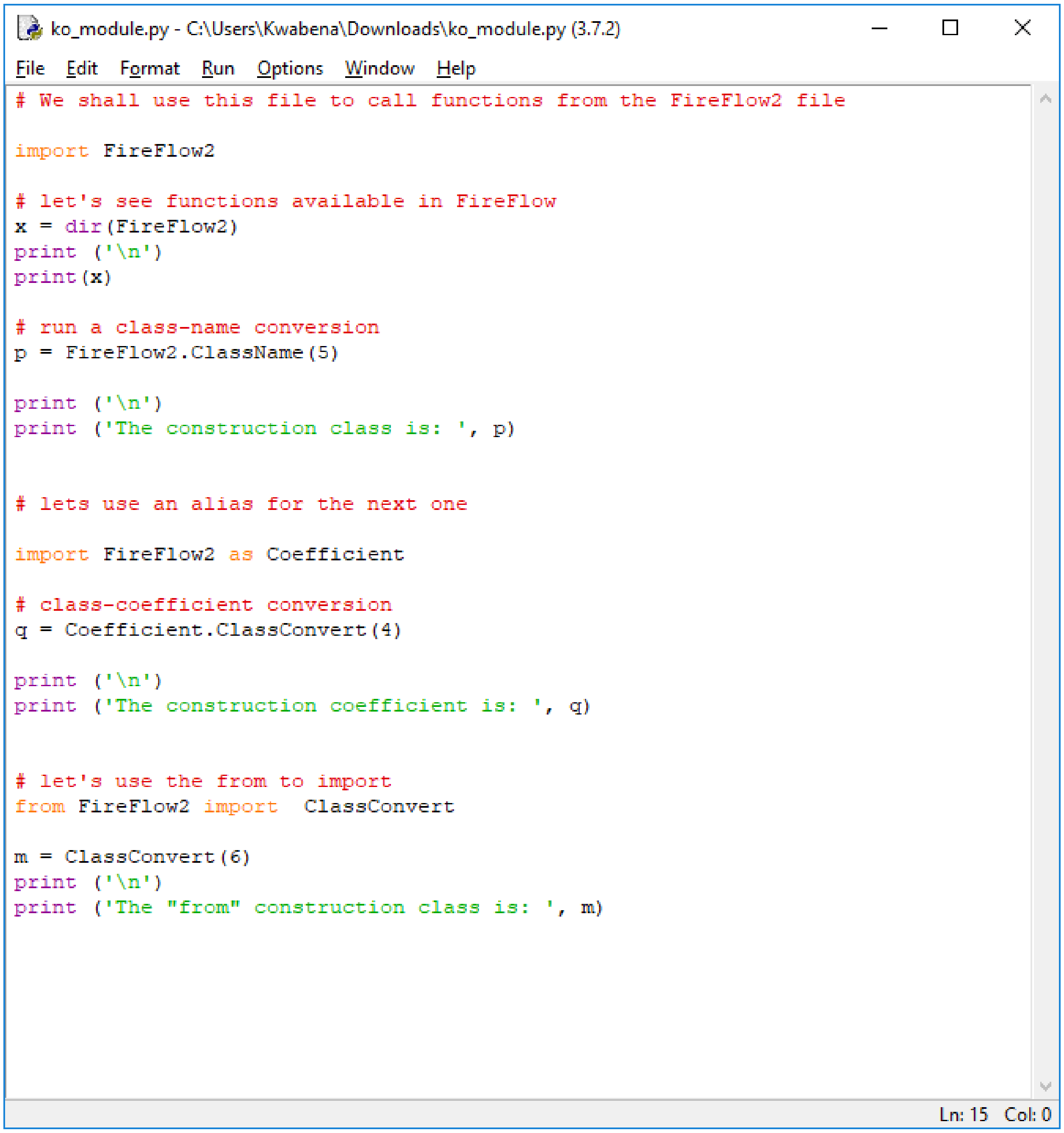
6.7 The *from* Keyword

The *import* in conjunction with the *from* keyword provide another means of importing parts – variables, functions, etc., from a module. In this method, the module file name is not prefixed to the function (or variable) name when it is called. The general form of the syntax is as follows.

```
from < filename > import  < functionname >

< var > = < functioname >(arguments)
```

Open a new session of IDLE (Python GUI).
Open the File Editor.

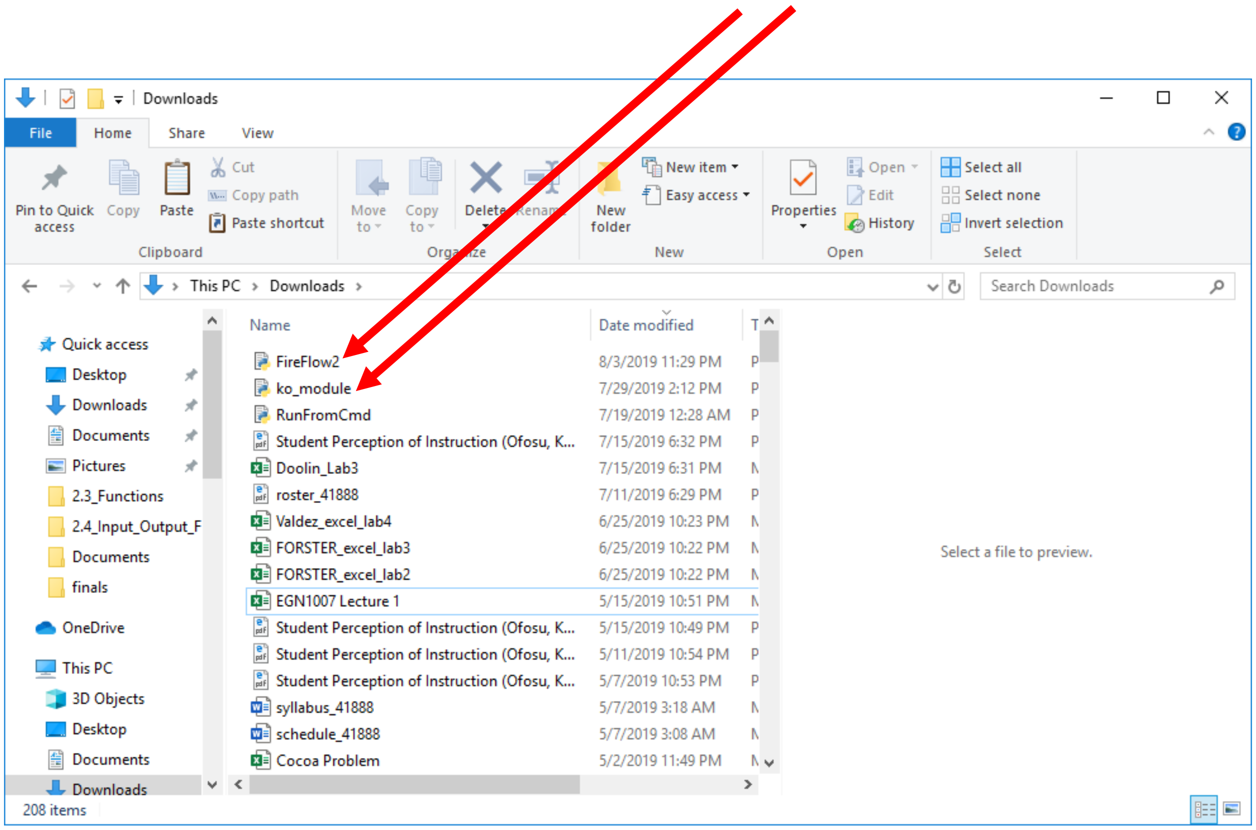
In the File Editor reproduce the following code.



ko_module.py - C:\Users\Kwabena\Downloads\ko_module.py (3.7.2)
File Edit Format Run Options Window Help
Ln: 15 Col: 0

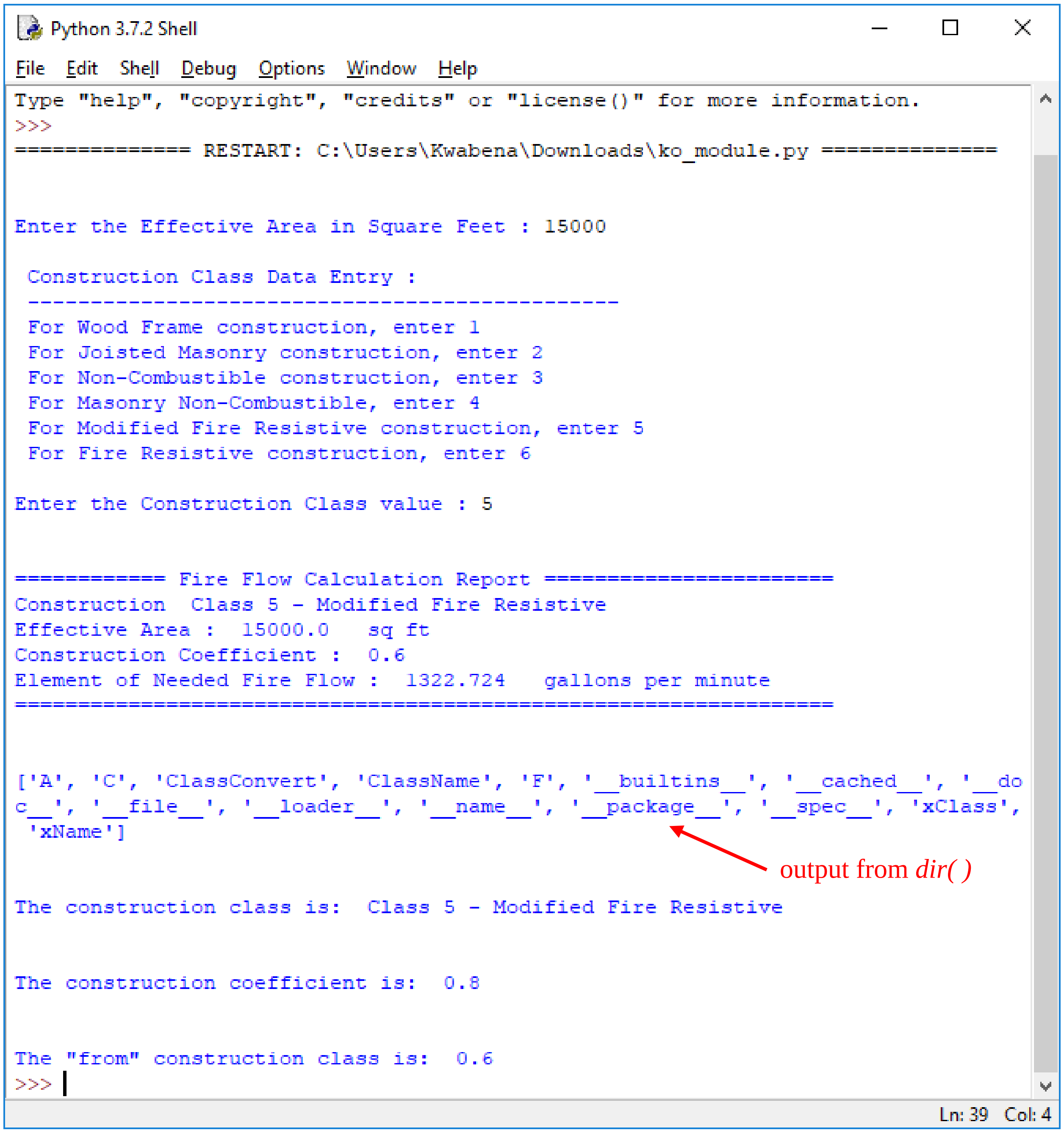
```
# We shall use this file to call functions from the FireFlow2 file

import FireFlow2

# let's see functions available in FireFlow
x = dir(FireFlow2)
print ('\n')
print(x)

# run a class-name conversion
p = FireFlow2.ClassName(5)

print ('\n')
print ('The construction class is: ', p)


# lets use an alias for the next one

import FireFlow2 as Coefficient

# class-coefficient conversion
q = Coefficient.ClassConvert(4)

print ('\n')
print ('The construction coefficient is: ', q)


# let's use the from to import
from FireFlow2 import  ClassConvert

m = ClassConvert(6)
print ('\n')
print ('The "from" construction class is: ', m)
```

Save your file.
This is your module file.
Close your module file.
Copy and paste your module file to your Downloads folder.
Locate your file named **FireFlow2** that was supplied with the course materials from Suncam.
Copy and Paste your **FireFlow2** file to your Downloads folder.



Name	Date modified
FireFlow2	8/3/2019 11:29 PM
ko_module	7/29/2019 2:12 PM
RunFromCmd	7/19/2019 12:28 AM
Student Perception of Instruction (Ofosu, K...	7/15/2019 6:32 PM
Doolin_Lab3	7/15/2019 6:31 PM
roster_41888	7/11/2019 6:29 PM
Valdez_excel_lab4	6/25/2019 10:23 PM
FORSTER_excel_lab3	6/25/2019 10:22 PM
FORSTER_excel_lab2	6/25/2019 10:22 PM
EGN1007 Lecture 1	5/15/2019 10:51 PM
Student Perception of Instruction (Ofosu, K...	5/15/2019 10:49 PM
Student Perception of Instruction (Ofosu, K...	5/11/2019 10:54 PM
Student Perception of Instruction (Ofosu, K...	5/7/2019 10:53 PM
syllabus_41888	5/7/2019 3:18 AM
schedule_41888	5/7/2019 3:08 AM
Cocoa Problem	5/2/2019 11:49 PM

Run the module file to run the module(s).



```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
============== RESTART: C:\Users\Kwabena\Downloads\ko_module.py ==============


Enter the Effective Area in Square Feet : 15000

 Construction Class Data Entry :
 -----------------------------------------------
 For Wood Frame construction, enter 1
 For Joisted Masonry construction, enter 2
 For Non-Combustible construction, enter 3
 For Masonry Non-Combustible, enter 4
 For Modified Fire Resistive construction, enter 5
 For Fire Resistive construction, enter 6

Enter the Construction Class value : 5


============ Fire Flow Calculation Report =======================
Construction  Class 5 - Modified Fire Resistive
Effective Area :  15000.0   sq ft
Construction Coefficient :  0.6
Element of Needed Fire Flow :  1322.724   gallons per minute
=================================================================


['A', 'C', 'ClassConvert', 'ClassName', 'F', '__builtins__', '__cached__', '__do
c__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'xClass',
 'xName']


The construction class is:  Class 5 - Modified Fire Resistive


The construction coefficient is:  0.8


The "from" construction class is:  0.6
>>> 
```

7. FILE HANDLING

7.1 Introduction

The built-in file handling functions are used for creating, reading, writing to, and deleting files.

7.2 The *open( )* Function

The main function for file handling is the *open( )* function which takes two arguments, namely *filename* and *mode*. The syntax is of the form

open(< filename > , < mode >)

The modes for opening a file are:

Mode	Name	Description
"r"	Read	Opens a file for reading. If the file does not exist, throws an error message.
"a"	Append	Opens a file to be appended. If the file does not exist, it creates it
"w"	Write	Opens a file for writing. Overwrites any existing content. If the file does not exist, it creates it
"x"	Create	Creates the file. If the file already exists, it throws an error message

In addition to the file opening modes, there are two text/ binary modes that specify whether the data should be handled as binary or text mode.

Binary/text Mode	Name	Description
“t”	Text	text mode.
“b”	Binary	binary mode, e.g. images etc.

The file opening modes and the text/binary modes may be used in combination. For example, unless otherwise specified, the “r” and the “t” are the respective defaults.

The syntax to open a file is of the form

< var > = open(“< path to file >” , “< mode(s) >”)

If the file and Python are in the same folder then just the name of the file will suffice, otherwise the full path to the file shall inserted. In either case, the name of the file shall include the relevant suffix, e.g. “.txt”. Also note that the arguments are entered within double quotation marks.

7.3 The *read( )* Method

The *read( )* method may now be called to read the contents of the file from the assigned < *var* >. For example:

print < var > . read( )

The read method will return the entire text in the file.

The following syntax may be used to limit the output returned.

print < var > . read(n)

where the text from the 1st line through the nth byte only, will be returned.

7.4 The *readline( )* Method

The *readline( )* method is used to read the contents line by line. Thus

```
print  < var > . readline(  )
```

will return the first line, whereas

```
print  < var > . read(  )
print  < var > . read(  )
```

will return the first two lines, and so on and so forth.

7.5 The *close( )* Method

The *close( )* method will close the file. For example:

```
< var > = open("< path to file >" ,  "rt")
print  < var > . readline( )
print  < var > . readline( )
:
:
:
print  < var > . readline( )
< var > . close()
```

7.6 The *write( )* Method

The syntax to open a file and append content is as of the form

< var > = open("< path to file >" , "a")
< var > . write(*"Additional Content")*
< var > . close( )

whereas the syntax to open a file and overwrite the content is of the form

< var > = open("< path to file >" , "w")
< var > . write(*"New content to replace existing")*
< var > . close( )

Note that both

< var > = open("< path to file >" , "a")

and

< var > = open("< path to file >" , "w")

will create a new file, if the file specified does not exist.

7.7 Deleting Files

The built-in os module has a function *os.remove( )* that is used to delete files.

The syntax is of the form

```
import os
os.remove("< path to file >")
```

If the file does not exist an error message will be thrown. To check that the file exists before deleting it, the following code can be used.

```
import os
if os.path.exists("< path to file >"):
        os.remove("< path to file >")
else:
         print("Cannot delete. This file does not exist")
```

7.8 Renaming Files

The syntax is of the form

```
import os
os.rename("< current file name >" , "< new file name >")
```

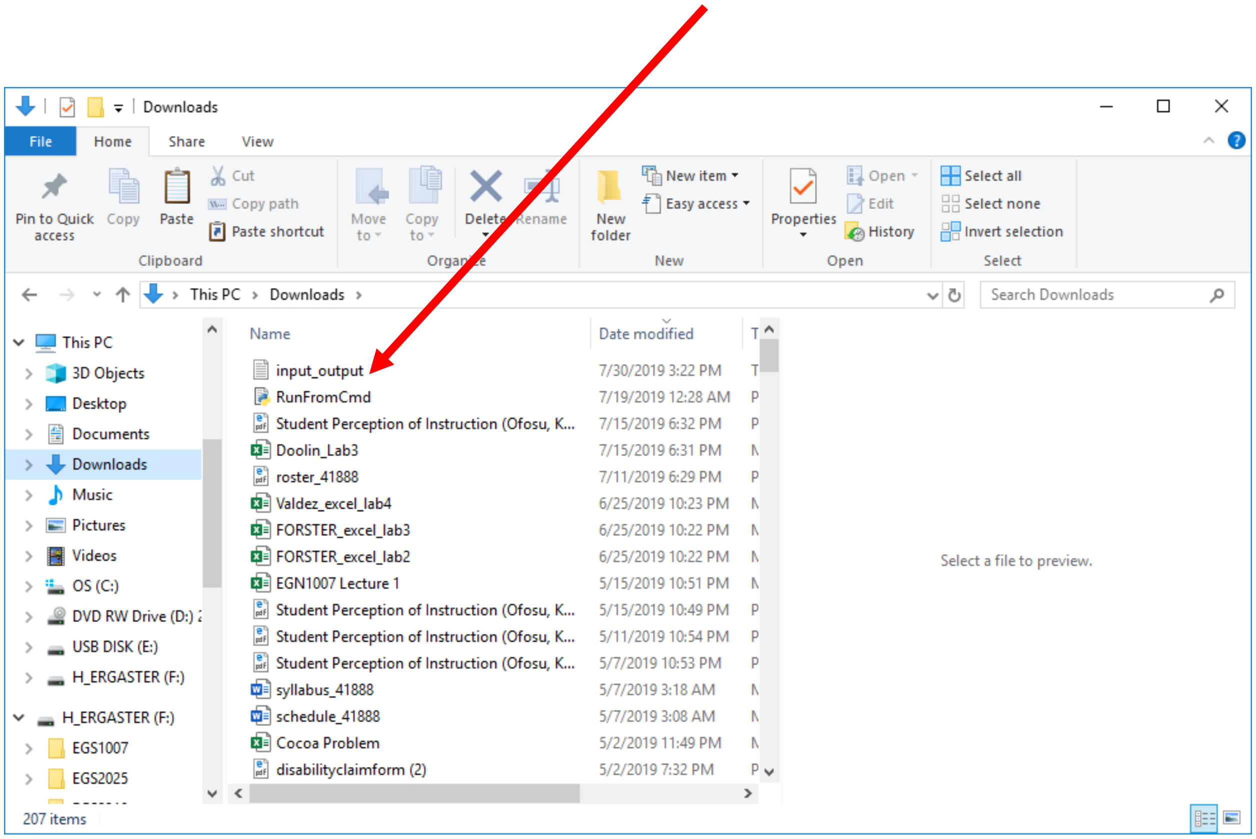
If the current file does not exist an error message will be thrown.

7.9 Deleting a Folder

The syntax is of the form

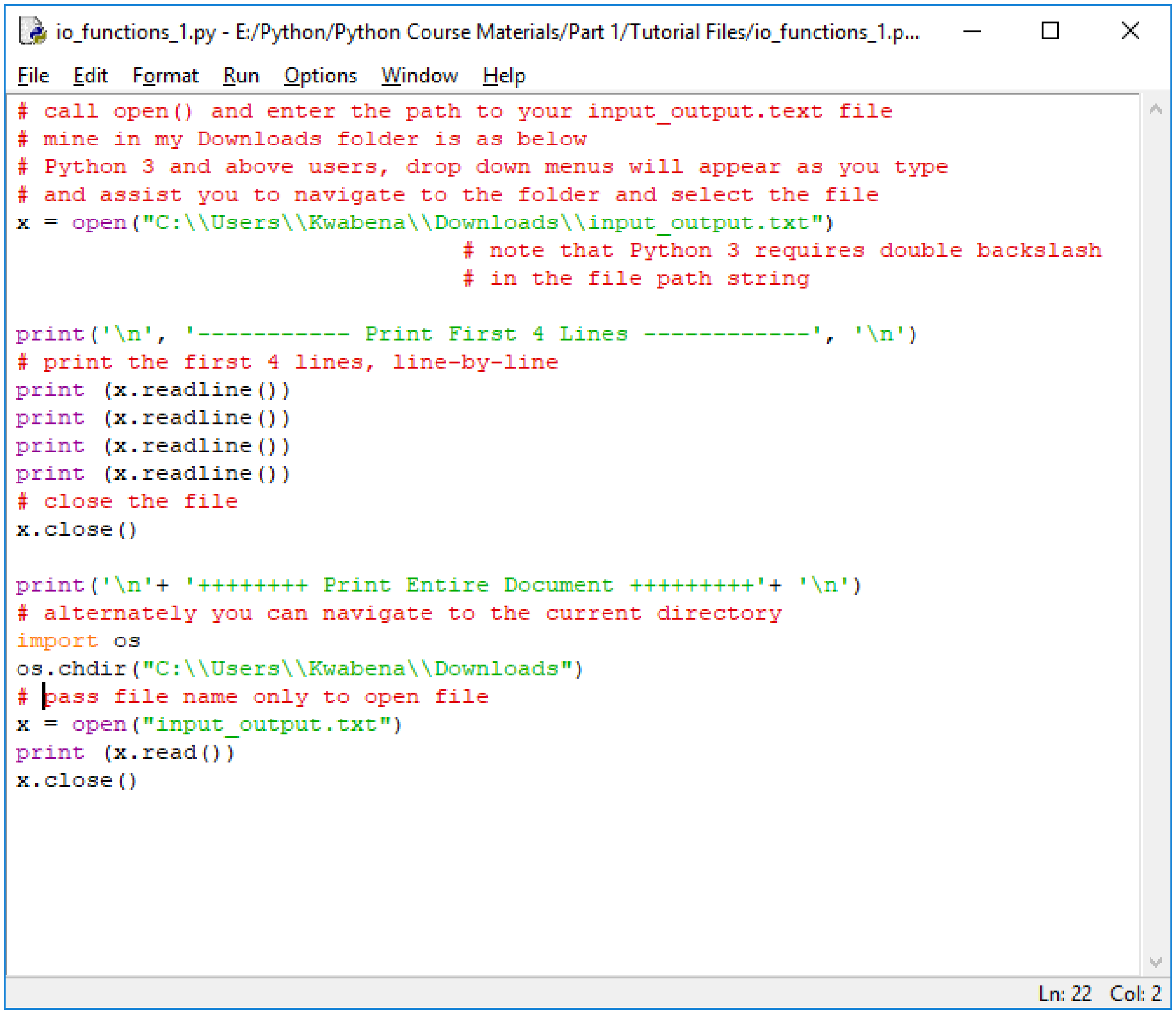
```
import os
os.remove("< path to folder >")
```

Open a new session of IDLE (Python GUI).
In your files supplied with this course from Suncam, locate the file called ***input_output.txt***.
Make a copy of your file inpu_output.txt and save it your Downloads folder.



Name	Date modified	T
input_output	7/30/2019 3:22 PM	T
RunFromCmd	7/19/2019 12:28 AM	P
Student Perception of Instruction (Ofosu, K...	7/15/2019 6:32 PM	P
Doolin_Lab3	7/15/2019 6:31 PM	N
roster_41888	7/11/2019 6:29 PM	P
Valdez_excel_lab4	6/25/2019 10:23 PM	N
FORSTER_excel_lab3	6/25/2019 10:22 PM	N
FORSTER_excel_lab2	6/25/2019 10:22 PM	N
EGN1007 Lecture 1	5/15/2019 10:51 PM	N
Student Perception of Instruction (Ofosu, K...	5/15/2019 10:49 PM	P
Student Perception of Instruction (Ofosu, K...	5/11/2019 10:54 PM	P
Student Perception of Instruction (Ofosu, K...	5/7/2019 10:53 PM	P
syllabus_41888	5/7/2019 3:18 AM	N
schedule_41888	5/7/2019 3:08 AM	N
Cocoa Problem	5/2/2019 11:49 PM	N
disabilityclaimform (2)	5/2/2019 7:32 PM	P

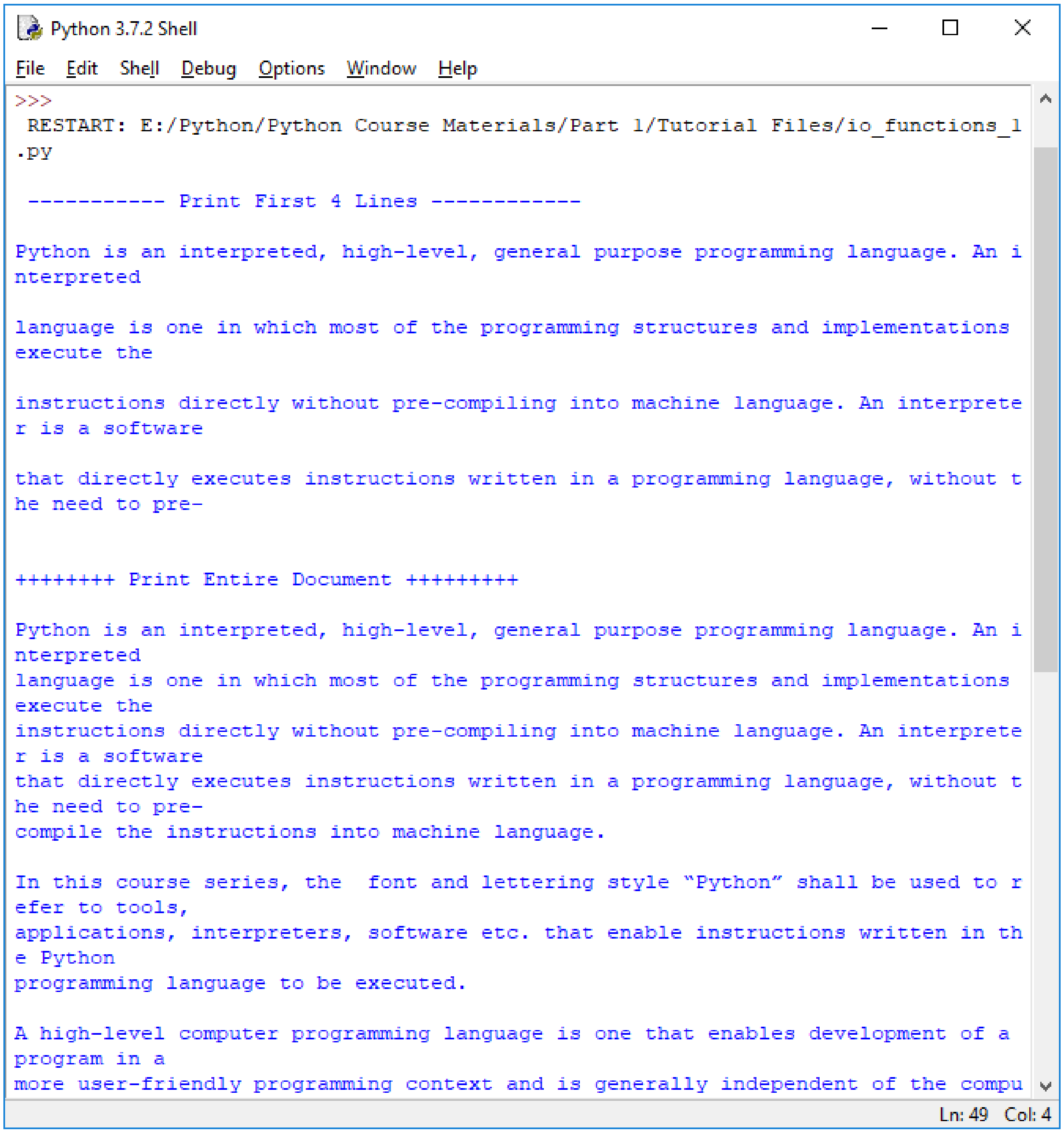
From IDLE (Python GUI), Open the File Editor.
In the File Editor reproduce the following code.
(Remember to type the path to **your** file on **your** computer).


io_functions_1.py - E:/Python/Python Course Materials/Part 1/Tutorial Files/io_functions_1.p...
File Edit Format Run Options Window Help
Ln: 22 Col: 2

```
# call open() and enter the path to your input_output.text file
# mine in my Downloads folder is as below
# Python 3 and above users, drop down menus will appear as you type
# and assist you to navigate to the folder and select the file
x = open("C:\\Users\\Kwabena\\Downloads\\input_output.txt")
                                # note that Python 3 requires double backslash
                                # in the file path string

print('\n', '----------- Print First 4 Lines ------------', '\n')
# print the first 4 lines, line-by-line
print (x.readline())
print (x.readline())
print (x.readline())
print (x.readline())
# close the file
x.close()

print('\n'+ '++++++++ Print Entire Document +++++++++'+ '\n')
# alternately you can navigate to the current directory
import os
os.chdir("C:\\Users\\Kwabena\\Downloads")
# pass file name only to open file
x = open("input_output.txt")
print (x.read())
x.close()
```

Save your file.
Run your file.

Review the results in IDLE.



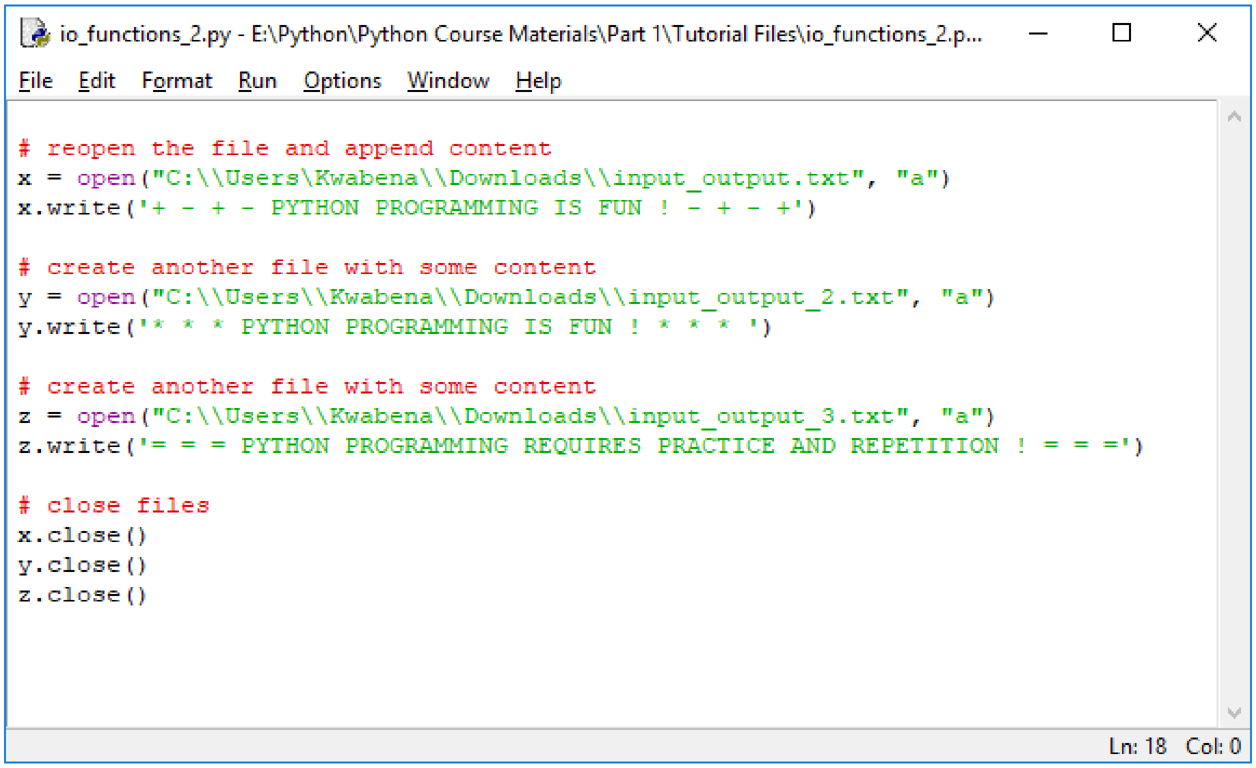
Python 3.7.2 Shell

File Edit Shell Debug Options Window Help

```
>>>
 RESTART: E:/Python/Python Course Materials/Part 1/Tutorial Files/io_functions_1
.py

 ----------- Print First 4 Lines ------------

Python is an interpreted, high-level, general purpose programming language. An i
nterpreted

language is one in which most of the programming structures and implementations
execute the

instructions directly without pre-compiling into machine language. An interprete
r is a software

that directly executes instructions written in a programming language, without t
he need to pre-


++++++++ Print Entire Document +++++++++

Python is an interpreted, high-level, general purpose programming language. An i
nterpreted
language is one in which most of the programming structures and implementations
execute the
instructions directly without pre-compiling into machine language. An interprete
r is a software
that directly executes instructions written in a programming language, without t
he need to pre-
compile the instructions into machine language.

In this course series, the  font and lettering style "Python" shall be used to r
efer to tools,
applications, interpreters, software etc. that enable instructions written in th
e Python
programming language to be executed.

A high-level computer programming language is one that enables development of a
program in a
more user-friendly programming context and is generally independent of the compu
```

Ln: 49 Col: 4

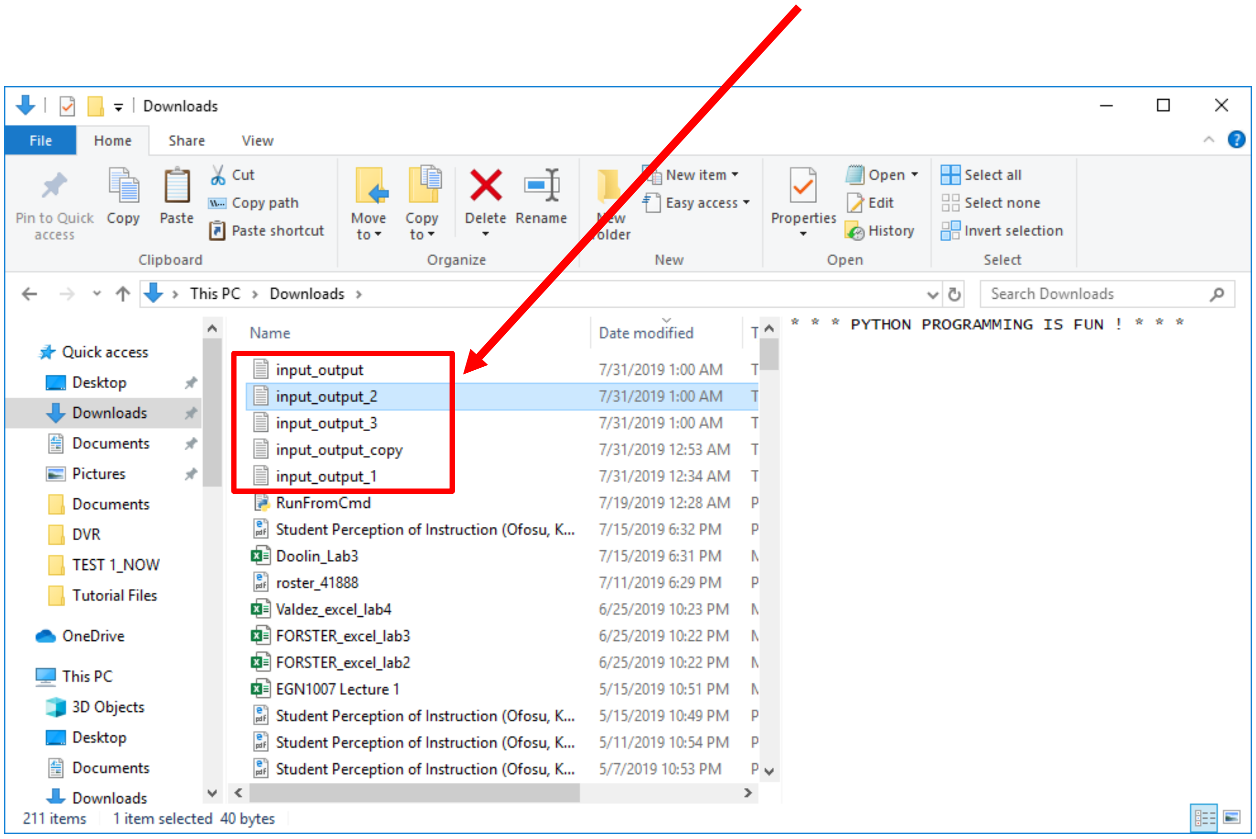
In the File Editor, Click on File.
Click on New File to open a new File Editor session.
Reproduce the following code.


io_functions_2.py - E:\Python\Python Course Materials\Part 1\Tutorial Files\io_functions_2.p...
File Edit Format Run Options Window Help
Ln: 18 Col: 0

```
# reopen the file and append content
x = open("C:\\Users\Kwabena\\Downloads\\input_output.txt", "a")
x.write('+ - + - PYTHON PROGRAMMING IS FUN ! - + - +')

# create another file with some content
y = open("C:\\Users\\Kwabena\\Downloads\\input_output_2.txt", "a")
y.write('* * * PYTHON PROGRAMMING IS FUN ! * * * ')

# create another file with some content
z = open("C:\\Users\\Kwabena\\Downloads\\input_output_3.txt", "a")
z.write('= = = PYTHON PROGRAMMING REQUIRES PRACTICE AND REPETITION ! = = =')

# close files
x.close()
y.close()
z.close()
```

Save your file.
Run your file.

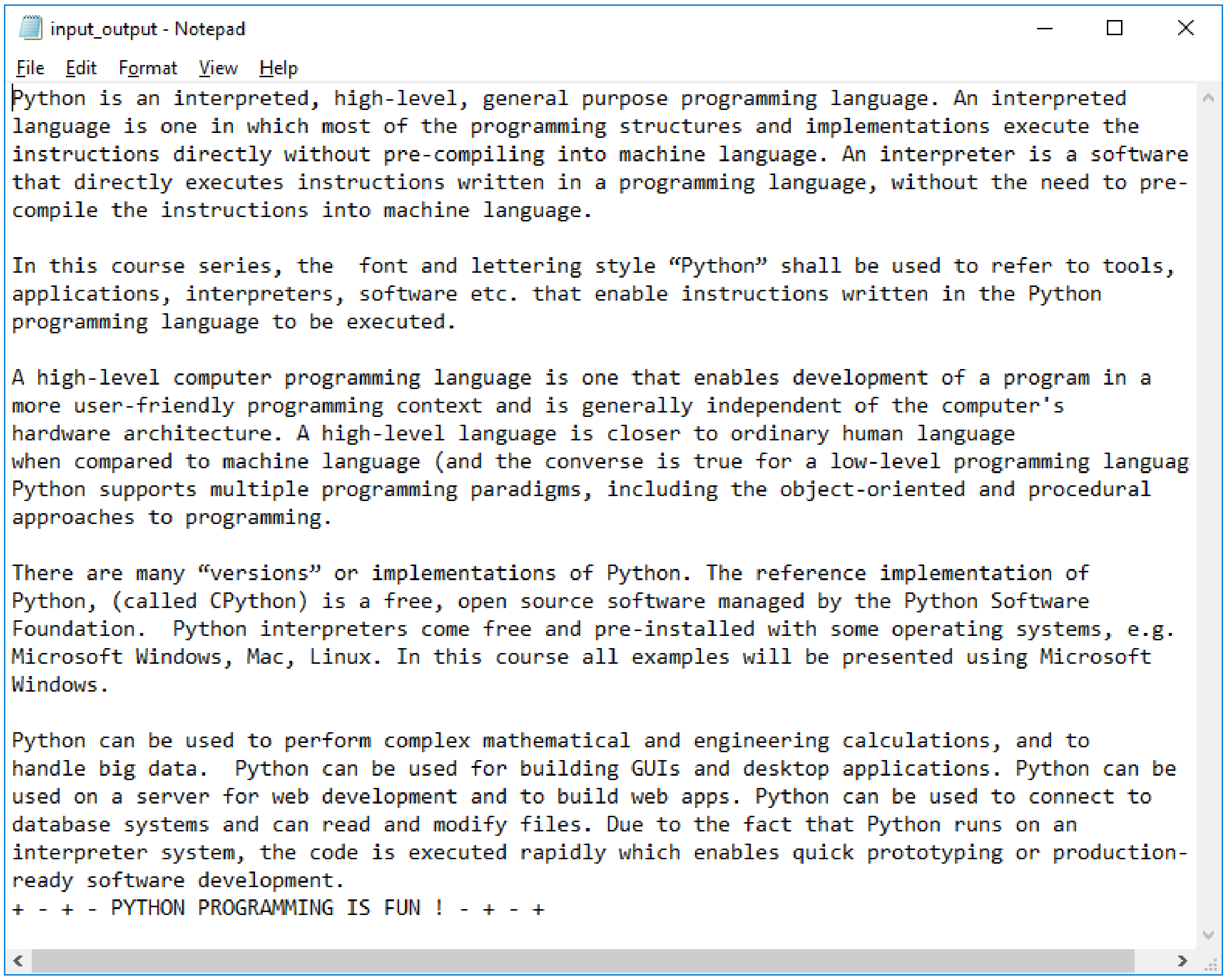
The new files are created in the directory.



Name	Date modified	T
input_output	7/31/2019 1:00 AM	T
input_output_2	7/31/2019 1:00 AM	T
input_output_3	7/31/2019 1:00 AM	T
input_output_copy	7/31/2019 12:53 AM	T
input_output_1	7/31/2019 12:34 AM	T
RunFromCmd	7/19/2019 12:28 AM	P
Student Perception of Instruction (Ofosu, K...	7/15/2019 6:32 PM	P
Doolin_Lab3	7/15/2019 6:31 PM	N
roster_41888	7/11/2019 6:29 PM	P
Valdez_excel_lab4	6/25/2019 10:23 PM	N
FORSTER_excel_lab3	6/25/2019 10:22 PM	N
FORSTER_excel_lab2	6/25/2019 10:22 PM	N
EGN1007 Lecture 1	5/15/2019 10:51 PM	N
Student Perception of Instruction (Ofosu, K...	5/15/2019 10:49 PM	P
Student Perception of Instruction (Ofosu, K...	5/11/2019 10:54 PM	P
Student Perception of Instruction (Ofosu, K...	5/7/2019 10:53 PM	P

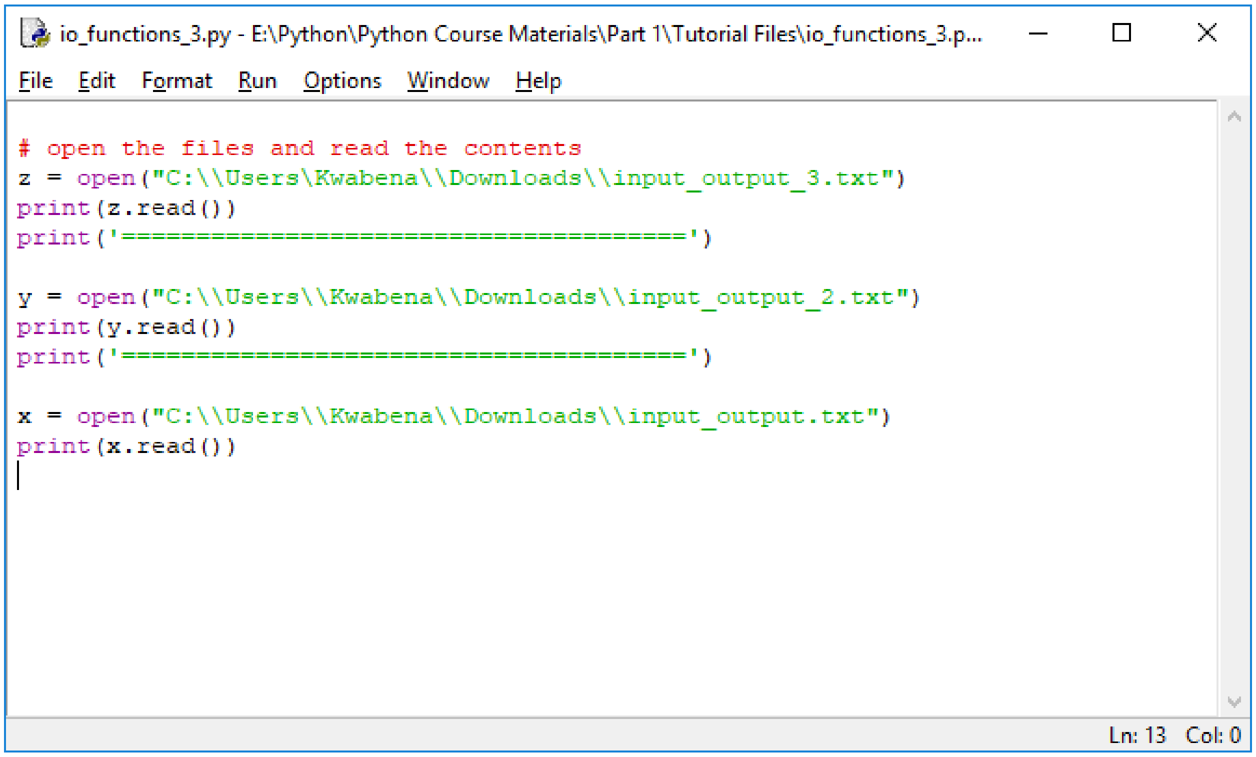
```
* * * PYTHON PROGRAMMING IS FUN ! * * *
```

Double click on ***input_output.txt*** to review the contents.



```
Python is an interpreted, high-level, general purpose programming language. An interpreted
language is one in which most of the programming structures and implementations execute the
instructions directly without pre-compiling into machine language. An interpreter is a software
that directly executes instructions written in a programming language, without the need to pre-
compile the instructions into machine language.

In this course series, the  font and lettering style “Python” shall be used to refer to tools,
applications, interpreters, software etc. that enable instructions written in the Python
programming language to be executed.

A high-level computer programming language is one that enables development of a program in a
more user-friendly programming context and is generally independent of the computer's
hardware architecture. A high-level language is closer to ordinary human language
when compared to machine language (and the converse is true for a low-level programming languag
Python supports multiple programming paradigms, including the object-oriented and procedural
approaches to programming.

There are many “versions” or implementations of Python. The reference implementation of
Python, (called CPython) is a free, open source software managed by the Python Software
Foundation.  Python interpreters come free and pre-installed with some operating systems, e.g.
Microsoft Windows, Mac, Linux. In this course all examples will be presented using Microsoft
Windows.

Python can be used to perform complex mathematical and engineering calculations, and to
handle big data.  Python can be used for building GUIs and desktop applications. Python can be
used on a server for web development and to build web apps. Python can be used to connect to
database systems and can read and modify files. Due to the fact that Python runs on an
interpreter system, the code is executed rapidly which enables quick prototyping or production-
ready software development.
+ - + - PYTHON PROGRAMMING IS FUN ! - + - +
```

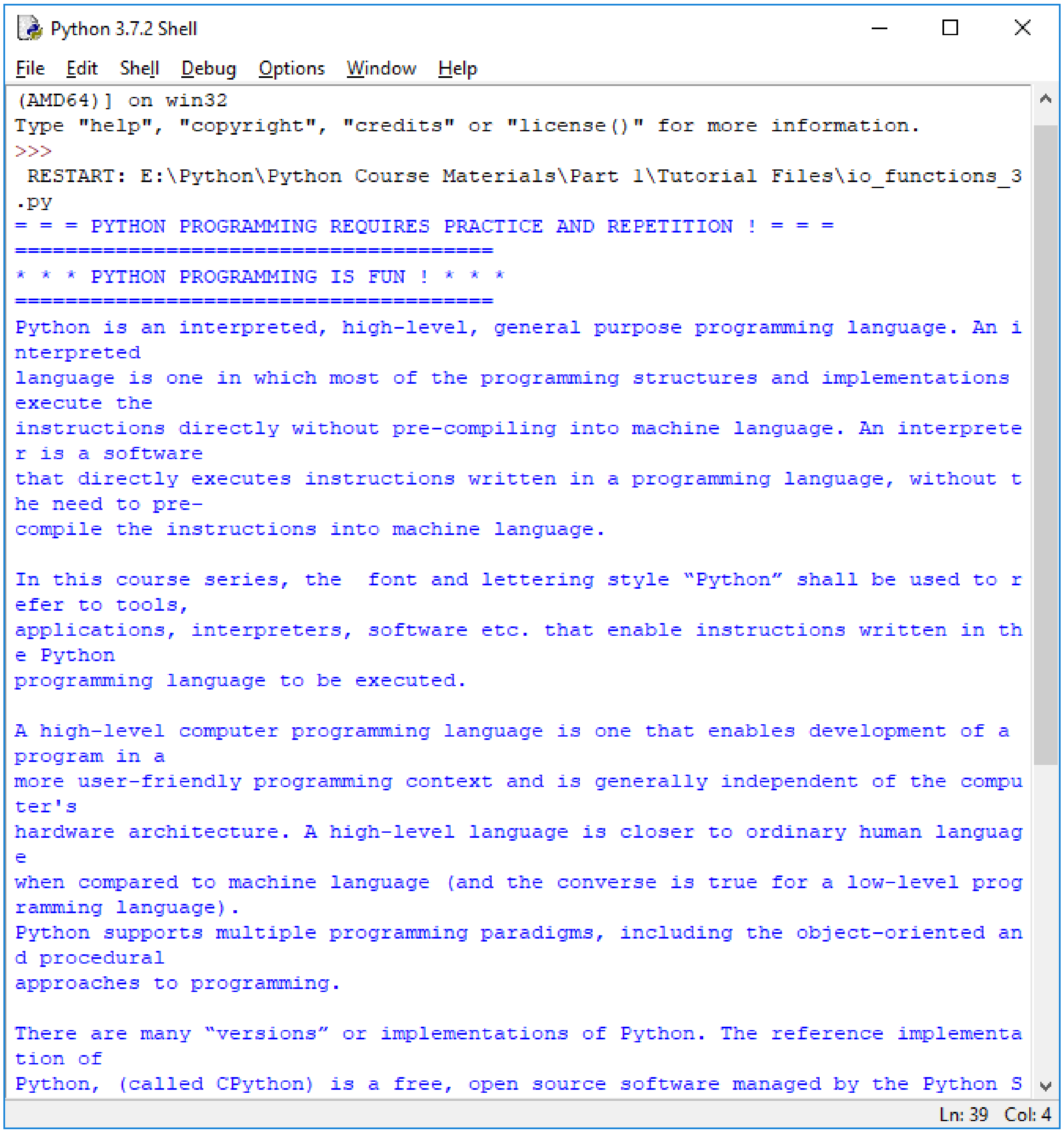
In the File Editor, Click on File.
Click on New File to open a new File Editor session.
Reproduce the following code.


io_functions_3.py - E:\Python\Python Course Materials\Part 1\Tutorial Files\io_functions_3.p...
File Edit Format Run Options Window Help
Ln: 13 Col: 0

```python
# open the files and read the contents
z = open("C:\\Users\Kwabena\\Downloads\\input_output_3.txt")
print(z.read())
print('======================================')

y = open("C:\\Users\\Kwabena\\Downloads\\input_output_2.txt")
print(y.read())
print('======================================')

x = open("C:\\Users\\Kwabena\\Downloads\\input_output.txt")
print(x.read())
```

Save your file.
Run your file.

Review the results in IDLE.

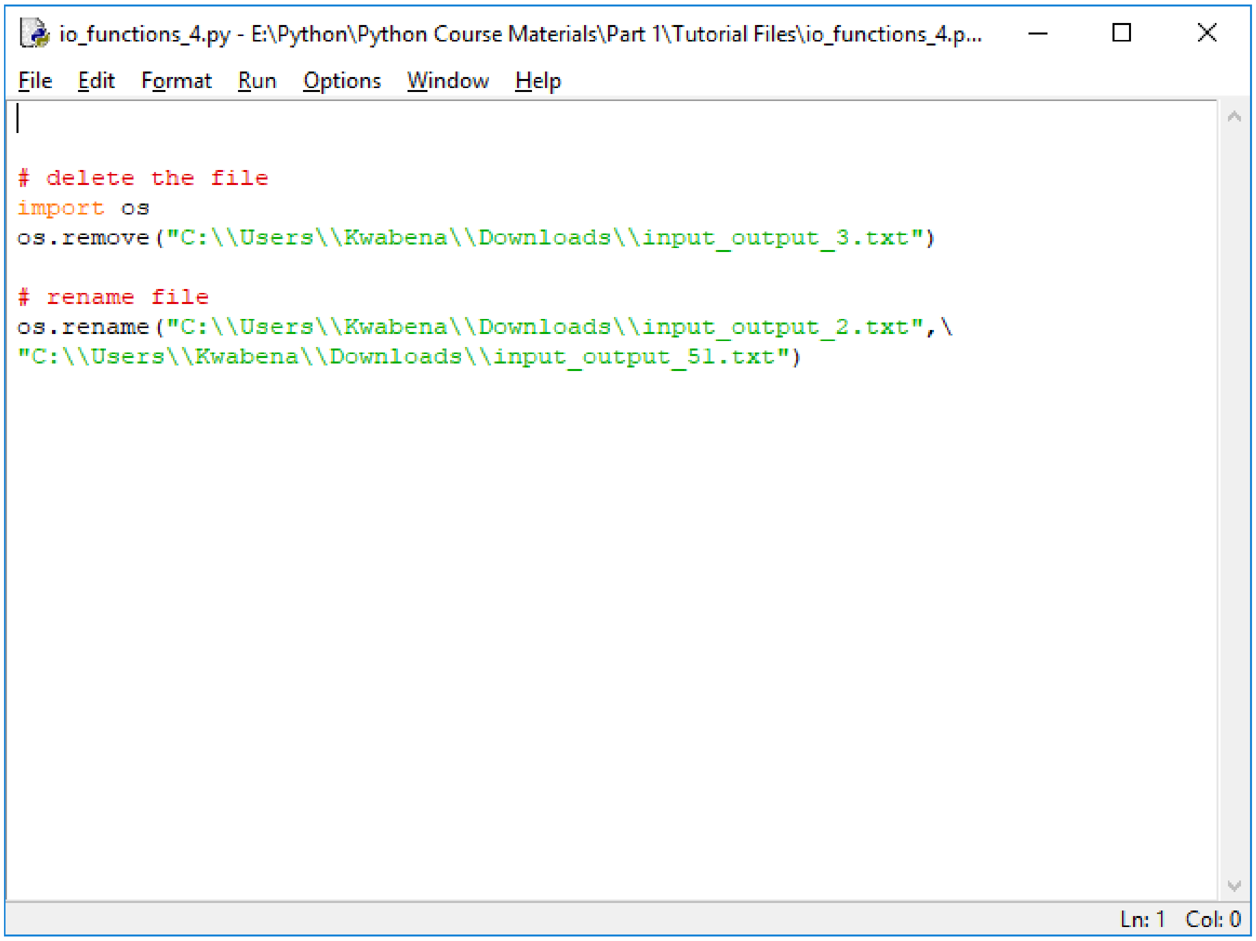

Python 3.7.2 Shell

File Edit Shell Debug Options Window Help

```
(AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: E:\Python\Python Course Materials\Part 1\Tutorial Files\io_functions_3
.py
= = = PYTHON PROGRAMMING REQUIRES PRACTICE AND REPETITION ! = = =
======================================
* * * PYTHON PROGRAMMING IS FUN ! * * *
======================================
Python is an interpreted, high-level, general purpose programming language. An i
nterpreted
language is one in which most of the programming structures and implementations
execute the
instructions directly without pre-compiling into machine language. An interprete
r is a software
that directly executes instructions written in a programming language, without t
he need to pre-
compile the instructions into machine language.

In this course series, the  font and lettering style “Python” shall be used to r
efer to tools,
applications, interpreters, software etc. that enable instructions written in th
e Python
programming language to be executed.

A high-level computer programming language is one that enables development of a
program in a
more user-friendly programming context and is generally independent of the compu
ter's
hardware architecture. A high-level language is closer to ordinary human languag
e
when compared to machine language (and the converse is true for a low-level prog
ramming language).
Python supports multiple programming paradigms, including the object-oriented an
d procedural
approaches to programming.

There are many “versions” or implementations of Python. The reference implementa
tion of
Python, (called CPython) is a free, open source software managed by the Python S
```

Ln: 39 Col: 4

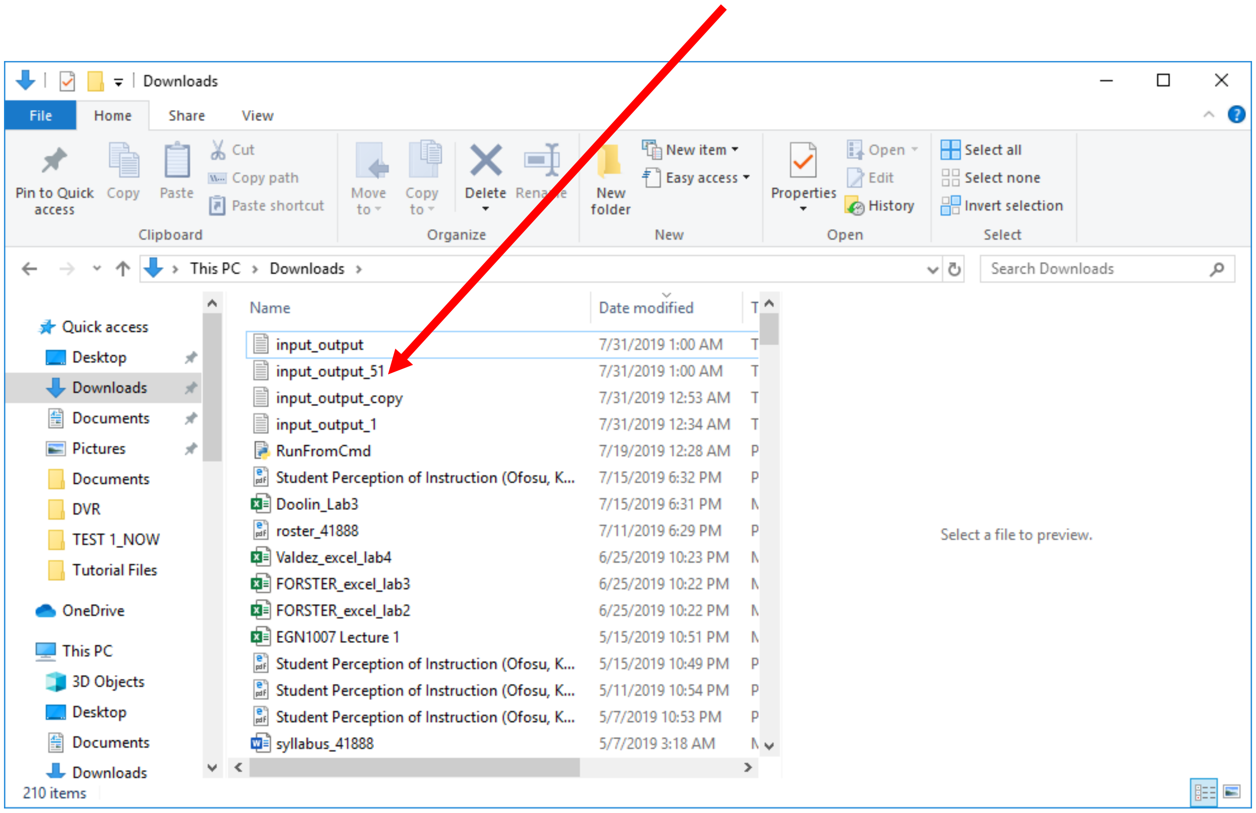
In the File Editor, Click on File.
Click on New File to open a new File Editor session.
Reproduce the following code.


io_functions_4.py - E:\Python\Python Course Materials\Part 1\Tutorial Files\io_functions_4.p...
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
# delete the file
import os
os.remove("C:\\Users\\Kwabena\\Downloads\\input_output_3.txt")

# rename file
os.rename("C:\\Users\\Kwabena\\Downloads\\input_output_2.txt",\
"C:\\Users\\Kwabena\\Downloads\\input_output_51.txt")
```

Save your file.
Run your file.

The file is deleted.
The file is renamed.



Name	Date modified
input_output	7/31/2019 1:00 AM
input_output_51	7/31/2019 1:00 AM
input_output_copy	7/31/2019 12:53 AM
input_output_1	7/31/2019 12:34 AM
RunFromCmd	7/19/2019 12:28 AM
Student Perception of Instruction (Ofosu, K...	7/15/2019 6:32 PM
Doolin_Lab3	7/15/2019 6:31 PM
roster_41888	7/11/2019 6:29 PM
Valdez_excel_lab4	6/25/2019 10:23 PM
FORSTER_excel_lab3	6/25/2019 10:22 PM
FORSTER_excel_lab2	6/25/2019 10:22 PM
EGN1007 Lecture 1	5/15/2019 10:51 PM
Student Perception of Instruction (Ofosu, K...	5/15/2019 10:49 PM
Student Perception of Instruction (Ofosu, K...	5/11/2019 10:54 PM
Student Perception of Instruction (Ofosu, K...	5/7/2019 10:53 PM
syllabus_41888	5/7/2019 3:18 AM

210 items

8. ERROR HANDLING AND EXCEPTIONS

8.1 Errors

It is common, even among seasoned programmers, that due to some error(s) in the code, the script may not work as expected, or it may run partially and prematurely or abruptly terminate or "crash," or not run at all. Identifying errors and addressing them is called **debugging**. In programming, **error handling** refers to techniques and practices used to test scripts and isolate errors. In *Python*, when an error occurs during execution, the interpreter will stop executing the code and return a built-in error message or **exception**. In error/exception handling, this is generally termed as, to **throw an exception**. In *Python* programming the preferred parlance is to **raise an exception**.

8.2 Types of Errors

Errors have several causes. **Syntax errors** are the result of any violation of the rules or syntax for coding the instructions in the programming language. For example, misspelled or omitted keywords, typos, incomplete branching or looping structures, inconsistent indenting, inadmissible use of mathematical operators, incorrect use of functions or function arguments, and many others.

A **run-time error** occurs during program execution when some value is processed, or some resource is accessed in a manner that is inadmissible to the programming language. This will cause the program execution to abruptly terminate or "crash." For example, dividing some value or variable by zero will cause an error as the value is mathematically indeterminate, or calling a variable or function that does not exist or does not exist yet.

Logic errors, commonly called **bugs**, occur when the program runs "normally," but produces unexpected or untenable results. In other words, upon review, the programmer knows that the results are incorrect, but from the point of view of the interpreter executing the program, the program is "fine." Logic errors may also exist if the script behaves erratically; for example, the results of a calculation are assigned to a different variable than intended. Due to the fact that the program will run "normally," there will be no exceptions thrown at the user. This makes logic errors particularly difficult to identify. The programmer must have some domain expertise of the underlying theories or mathematical models being implemented in the program. The program must be tested repeatedly, and the results thoroughly scrutinized, verified, and validated.

Some common causes of logic errors include:

- omission of relevant code
- incorrect sequence of instructions
- calling the wrong variables or functions
- incorrect choice of branching and looping structures
- incorrect variables and/ or logic in conditional statements
- incorrect use of (loop) variables in loops
- incorrect referencing to collections' indices

8.3 Handling Exceptions in *Python*

During program execution, if an error occurs, the interpreter will return the applicable built-in error message (exception). Ordinarily, a non-expert end-user will not know what the exception means or what steps should be taken to address it. This will make the program appear confounding and quite unprofessional from the end-user's point of view, but more seriously, for a web application, it will open up and expose the program, the web page(s), and the server to security threats. It is therefore the programmer's responsibility to anticipate potential errors and add code that will address them in such a manner that the exceptions will not be raised to the end-user. This is the basis of error/ exception handling. Errors that are anticipated and addressed such that the relevant exception does not appear to the end-user, are referred to as **handled errors**, otherwise they are referred to as **unhandled errors**.

Examples of Python exceptions include:

Exception	**Raised when**
ImportError	an import module is not found.
IndexError	an index of a sequence is out of range
KeyError	when a referenced key does not exist in a dictionary type.
KeyboardInterrupt	when the user hits interrupt on the keyboard i.e. **Ctrl** + **c** or **Ctrl** + **Del**
MemoryError	an operation runs out of memory.
NameError	a variable does not exist in either a local or global scope
OverflowError	an arithmetic operation returns a result too large to be represented
RuntimeError	an error does not fall under any other category
SyntaxError	the parser finds a syntax error
IndentationError	the indentation is incorrect
TabError	tabs and spaces used to implement indentation are inconsistent
TypeError	an operation or function is applied to an incorrect data type
ValueError	a function gets argument of correct data type but improper value
ZeroDivisionError	the divisor in division or modulo is zero

Please review the *Python* literature for a comprehensive list and description of the *Python* built-in exceptions.

8.4 The *try … except* Clause

The simplest form of error handling in *Python* is the *try…except* structure. The syntax is of the form

```
try:
…       < try block >
except:
…       < except block >
```

If an exception will be thrown due to the <*try block*>, it will be handled by the <*except block*>. In other words, if an error occurs in the <*try block*>, the end user will not see the exception **raised** by Python but will experience whatever is done under the <*except block*>. Or to put it in yet another way, if an error occurs in the <*try block*>, code execution is immediately transferred to the <*except block*>.

8.5 The *try … except* Clause for Multiple Exceptions

If specific exceptions are anticipated, as many *except* blocks as desired may be added to handle those specific exceptions. The syntax is of the form

```
try:
…       < try block >
except <Name of exception 1 > :
…       < except block  1>
except <Name of exception 2 > :
…       < except block 2 >
except <Name of exception 3 > :
…       < except block 3 >
:
:
:
except <Name of exception x > :
…       < except block x >
```

```
except:
…       < except block >  # if any other exception is raised in the try block
                          # apart from any  of the above,
                          # use this except block
```

8.6 The *else* Keyword

The *else* keyword can be incorporated in the *try …except* handler to designate a block of code to execute if no exceptions are raised in the *try* block. The syntax is of the form

```
try:
…       < try block >
except <Name of exception > :
…       < except block >
:
:
except <Name of exception > :
…       < except block >
else:
…       < else block >  # if no exceptions raised in the try block
                        # apply use this else block
```

8.7 The *finally* Block

A *finally* block incorporated in the *try … except* handler will execute that block of code regardless of whether an exception is raised in the *try* block or not. The syntax is of the form

```
try:
…       < try block >
except :
…       < except block >
:
```

```
:
:
finally :
...       < finally block >  # this block runs regardless of whether
                             # exceptions raised in the try block or not
```

8.8 The *raise* Keyword

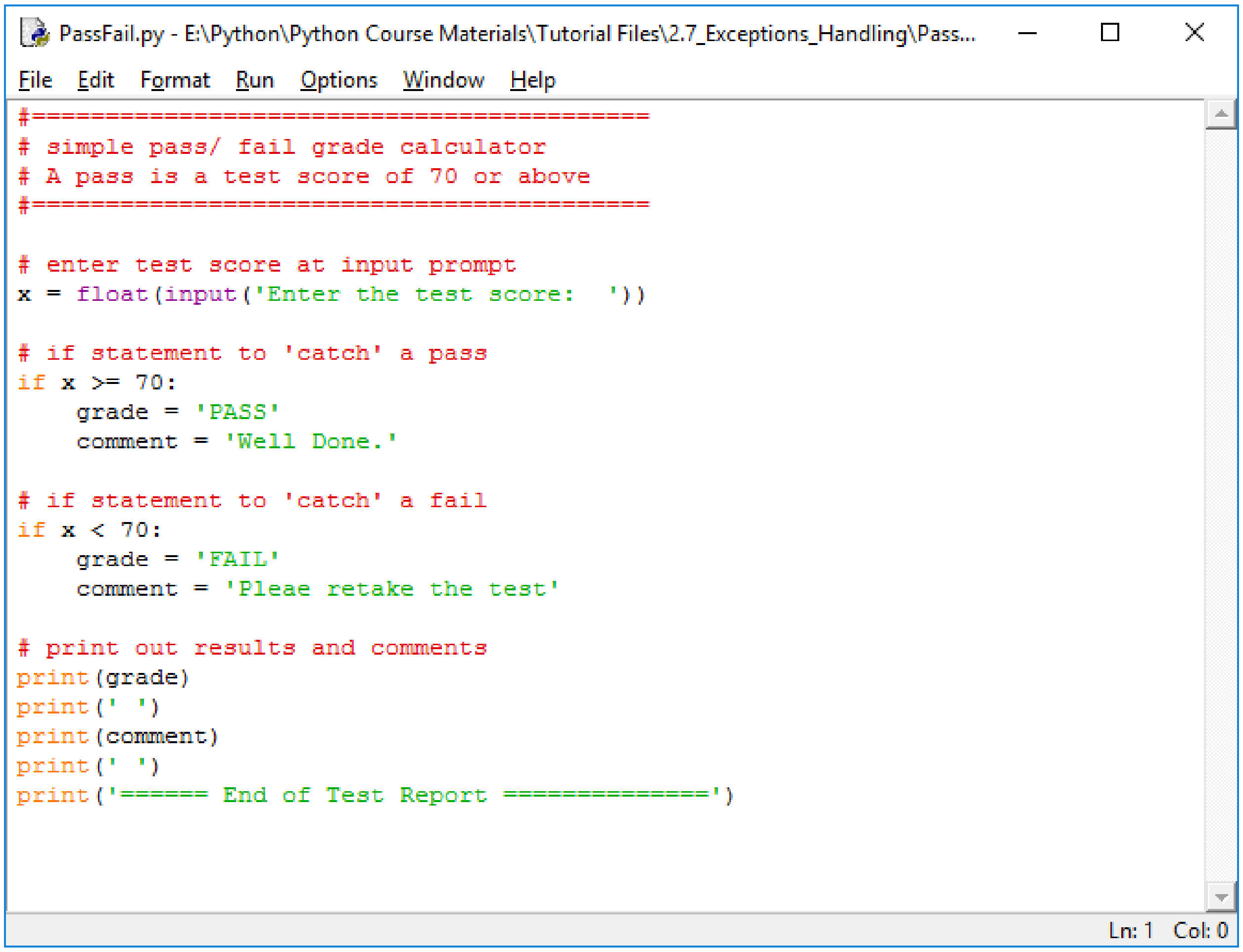
The *raise* clause in conjunction with an *if* statement can be used to generate a custom exception. The syntax is of the form

```
if (< conditional statement >) :
...       raise Exception (< custom statement >)
```

If the conditional statement returns the Boolean value of "true", then a customized exception that includes a concatenated string of the keyword *Exception* and the < *custom statement* > will be thrown at the user. The < *custom statement* > may incorporate variables, and other data types as needed (by using string functions or methods).

Open a new session of IDLE (Python GUI).
Open the File Editor.
In the File Editor, reproduce or copy over the code for the Pass/Fail Grade Calculator app that was developed in Chapter 4 of this course.

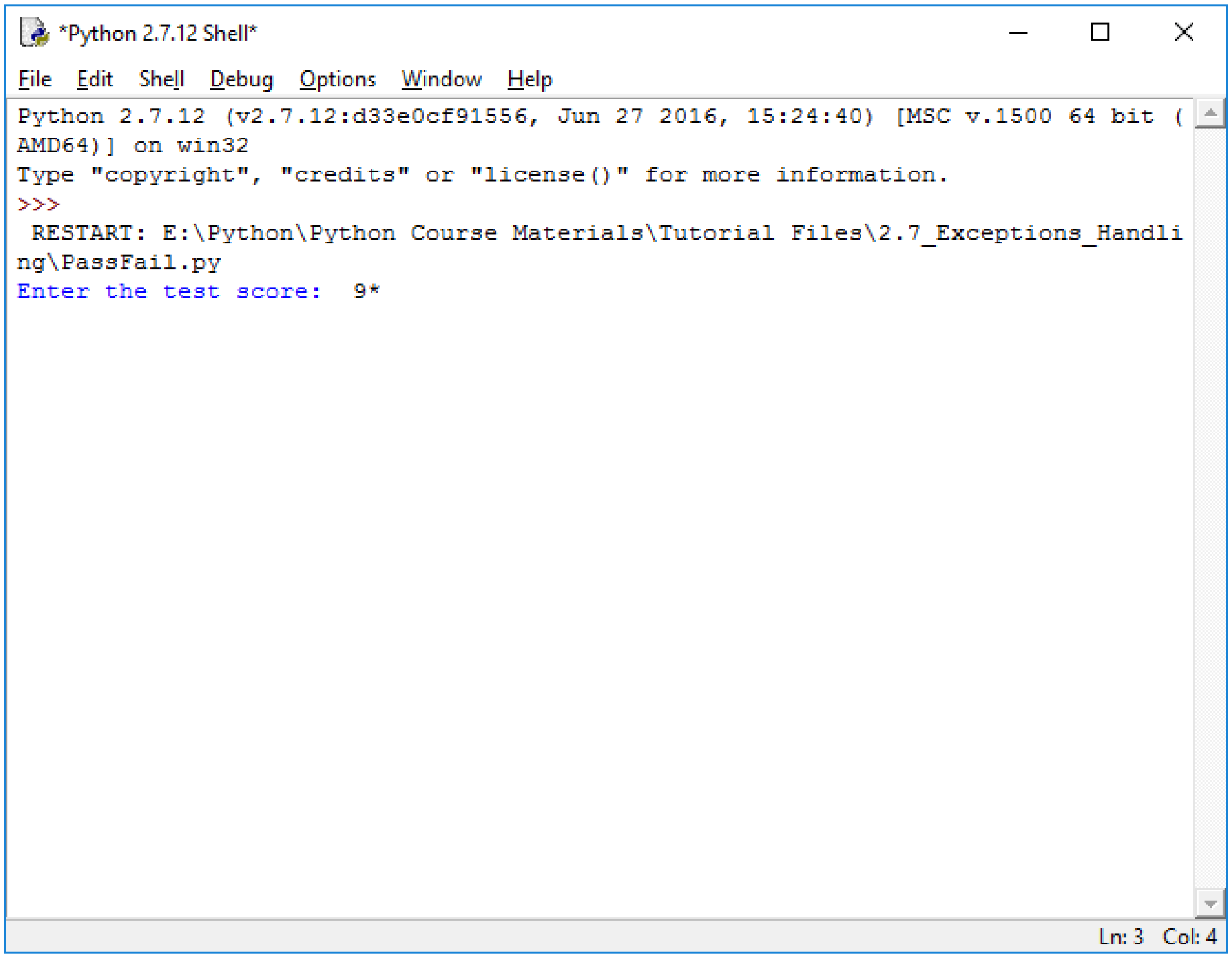
The code is as follows.



PassFail.py - E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling\Pass...

```
#===========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#===========================================

# enter test score at input prompt
x = float(input('Enter the test score:  '))

# if statement to 'catch' a pass
if x >= 70:
    grade = 'PASS'
    comment = 'Well Done.'

# if statement to 'catch' a fail
if x < 70:
    grade = 'FAIL'
    comment = 'Pleae retake the test'

# print out results and comments
print(grade)
print(' ')
print(comment)
print(' ')
print('====== End of Test Report ==============')
```

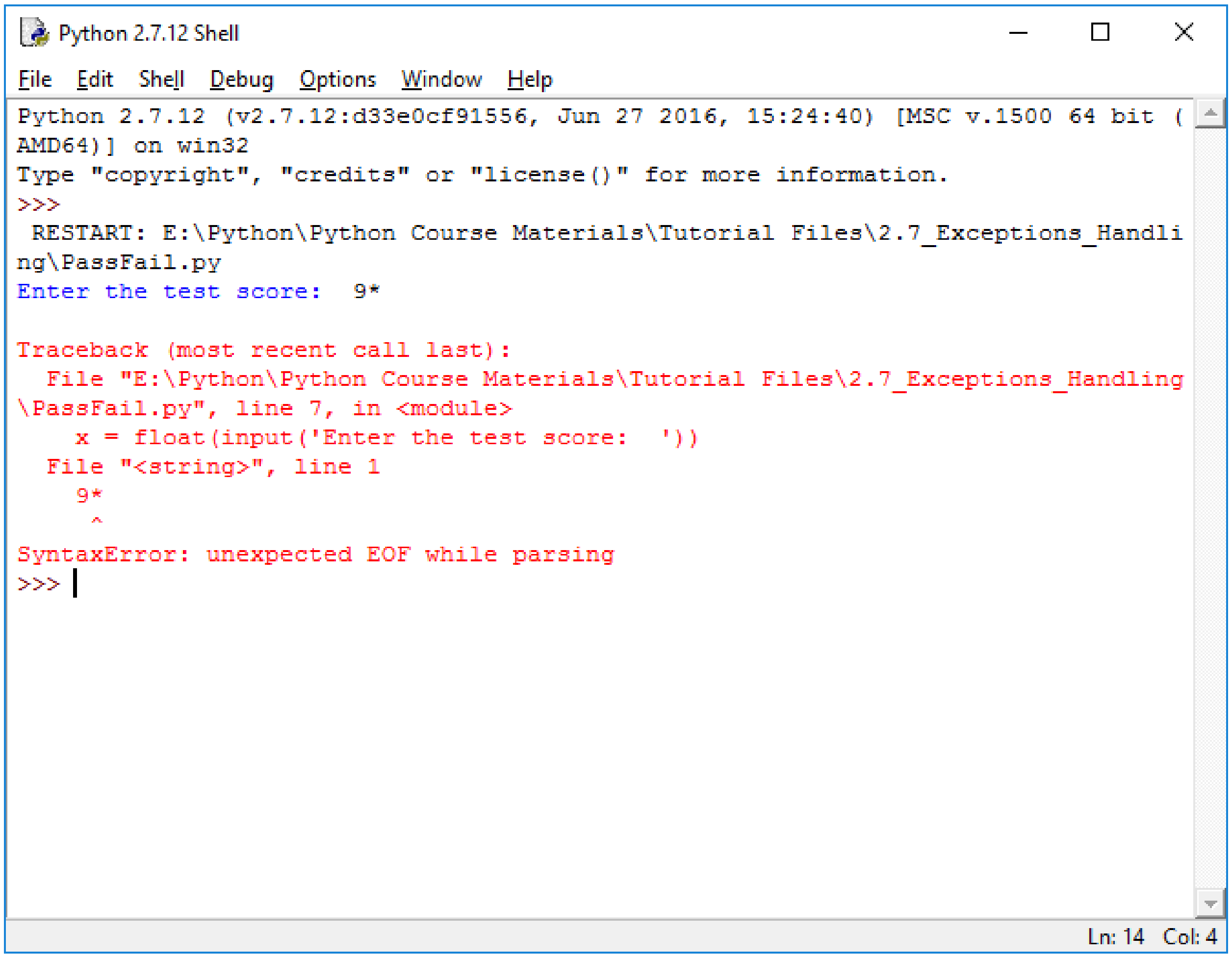
Run the file.
When prompted, enter the intentionally erroneous grade of "9*".



```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail.py
Enter the test score:  9*
```

Hit **Enter** to continue code execution.

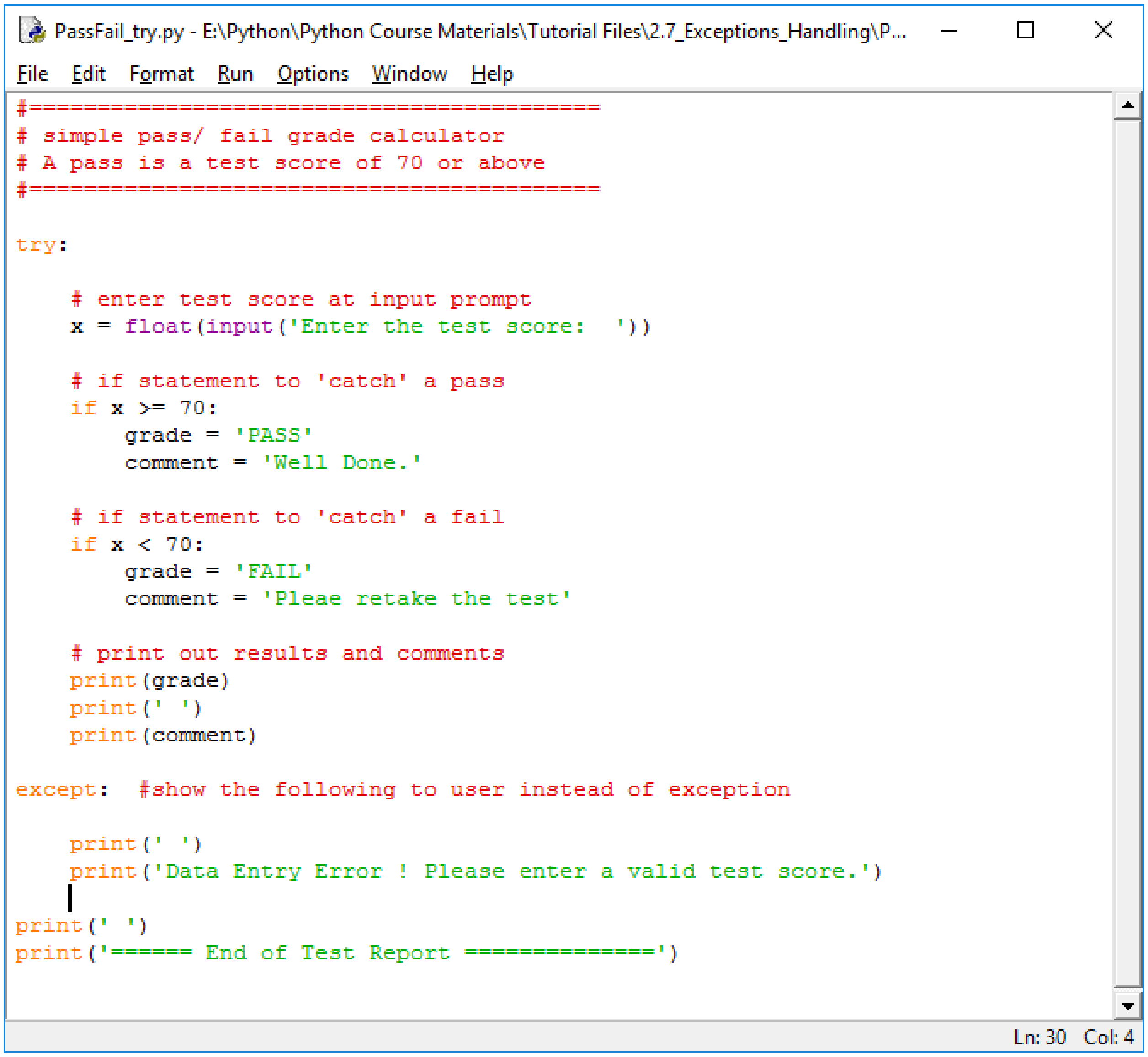
An exception is raised.



```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail.py
Enter the test score:  9*

Traceback (most recent call last):
  File "E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling
\PassFail.py", line 7, in <module>
    x = float(input('Enter the test score:  '))
  File "<string>", line 1
    9*
     ^
SyntaxError: unexpected EOF while parsing
>>> 
```

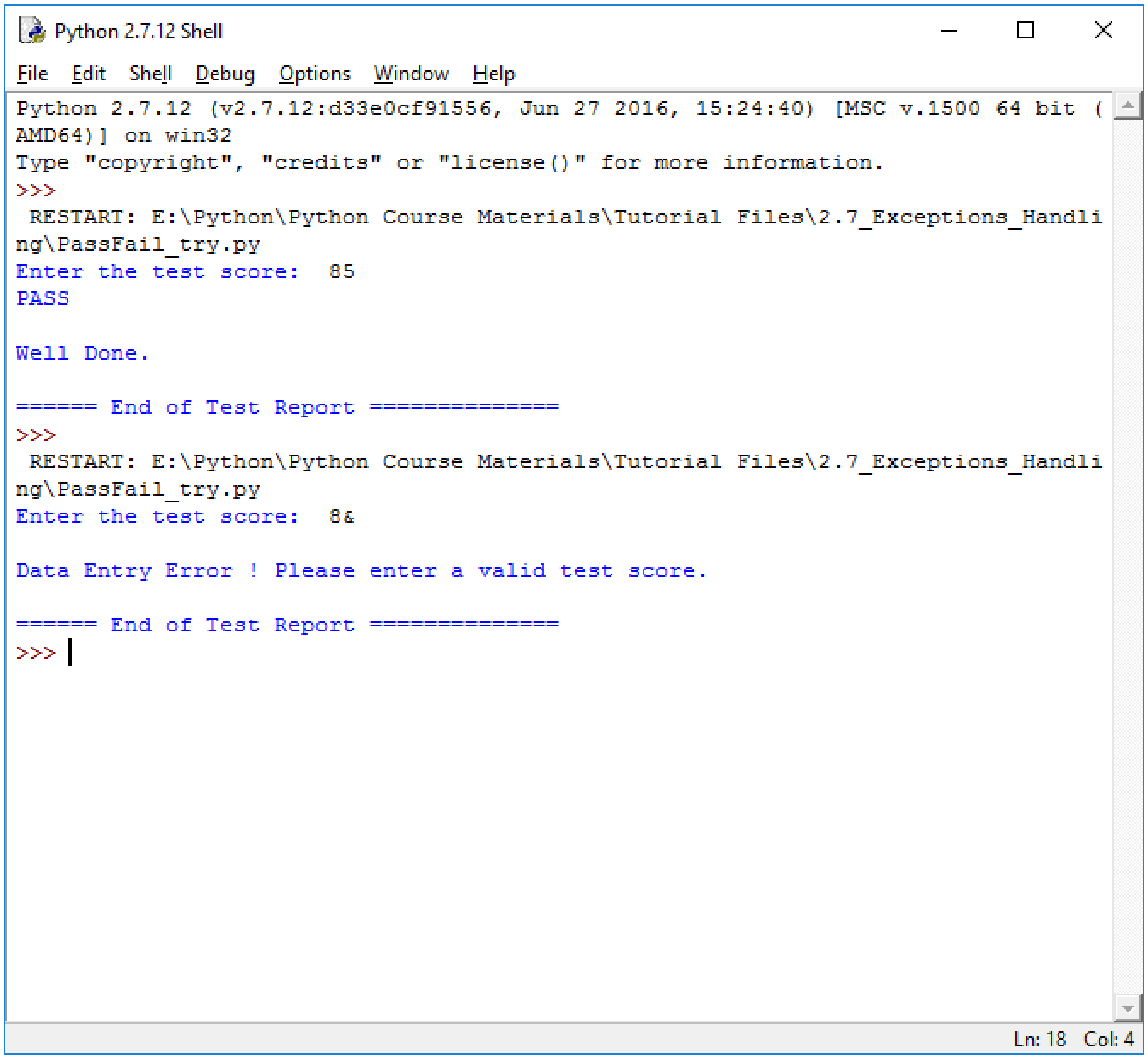
The “non-numeric” inadmissible value entered cannot be converted to a float. Thus the program crashes.

Granted the programmer can anticipate that an end-user may make this type of data entry error, a *try … except* handler can be used to “catch” such an error as follows.


PassFail_try.py - E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling\P...
File Edit Format Run Options Window Help
Ln: 30 Col: 4

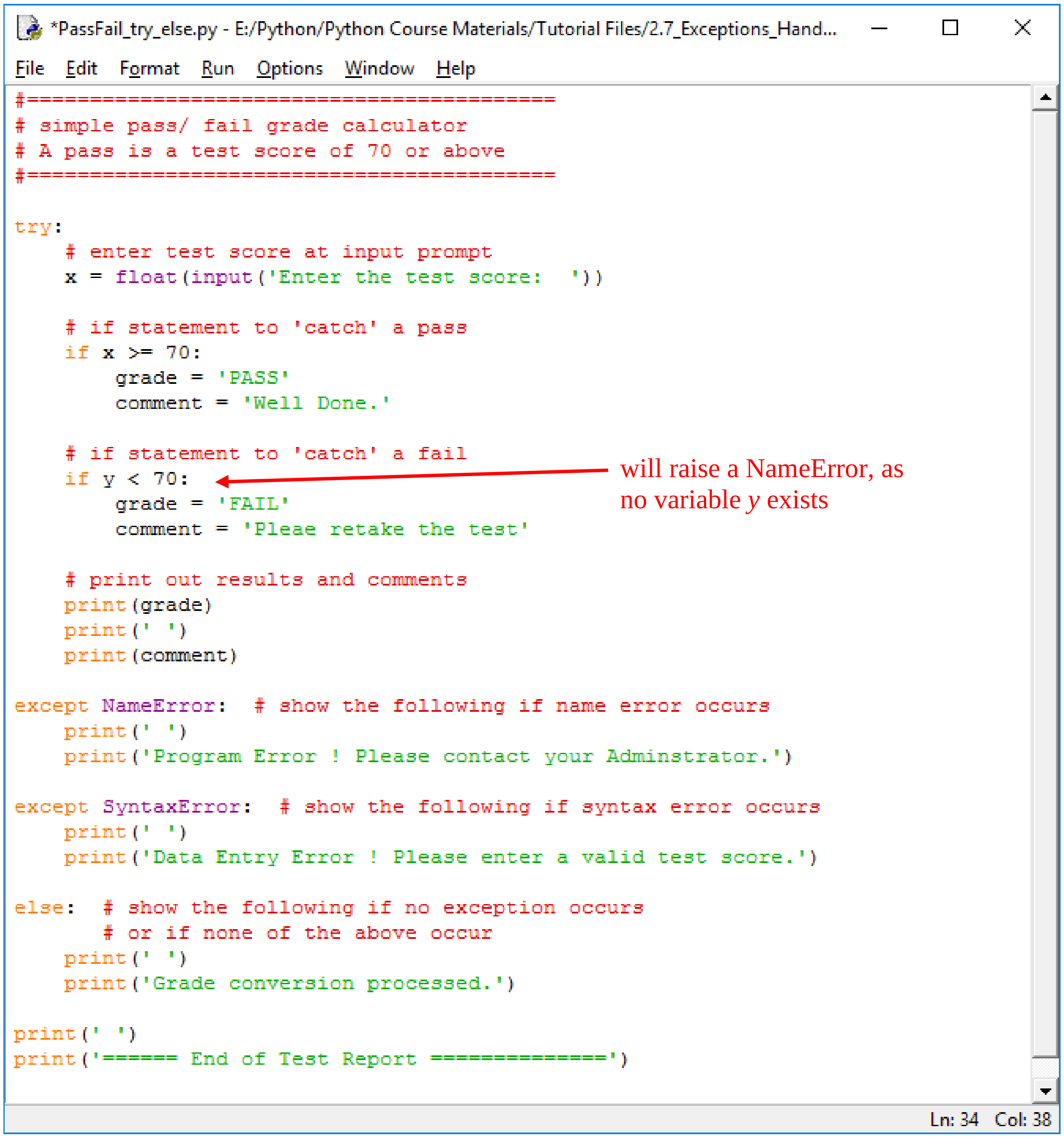
```
#==========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#==========================================

try:

    # enter test score at input prompt
    x = float(input('Enter the test score:  '))

    # if statement to 'catch' a pass
    if x >= 70:
        grade = 'PASS'
        comment = 'Well Done.'

    # if statement to 'catch' a fail
    if x < 70:
        grade = 'FAIL'
        comment = 'Pleae retake the test'

    # print out results and comments
    print(grade)
    print(' ')
    print(comment)

except:  #show the following to user instead of exception

    print(' ')
    print('Data Entry Error ! Please enter a valid test score.')

print(' ')
print('====== End of Test Report ==============')
```

Run the file using a valid test score.
Run the file using a "data entry error."



```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail_try.py
Enter the test score:  85
PASS

Well Done.

====== End of Test Report ==============
>>>
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail_try.py
Enter the test score:  8&

Data Entry Error ! Please enter a valid test score.

====== End of Test Report ==============
>>> 
```

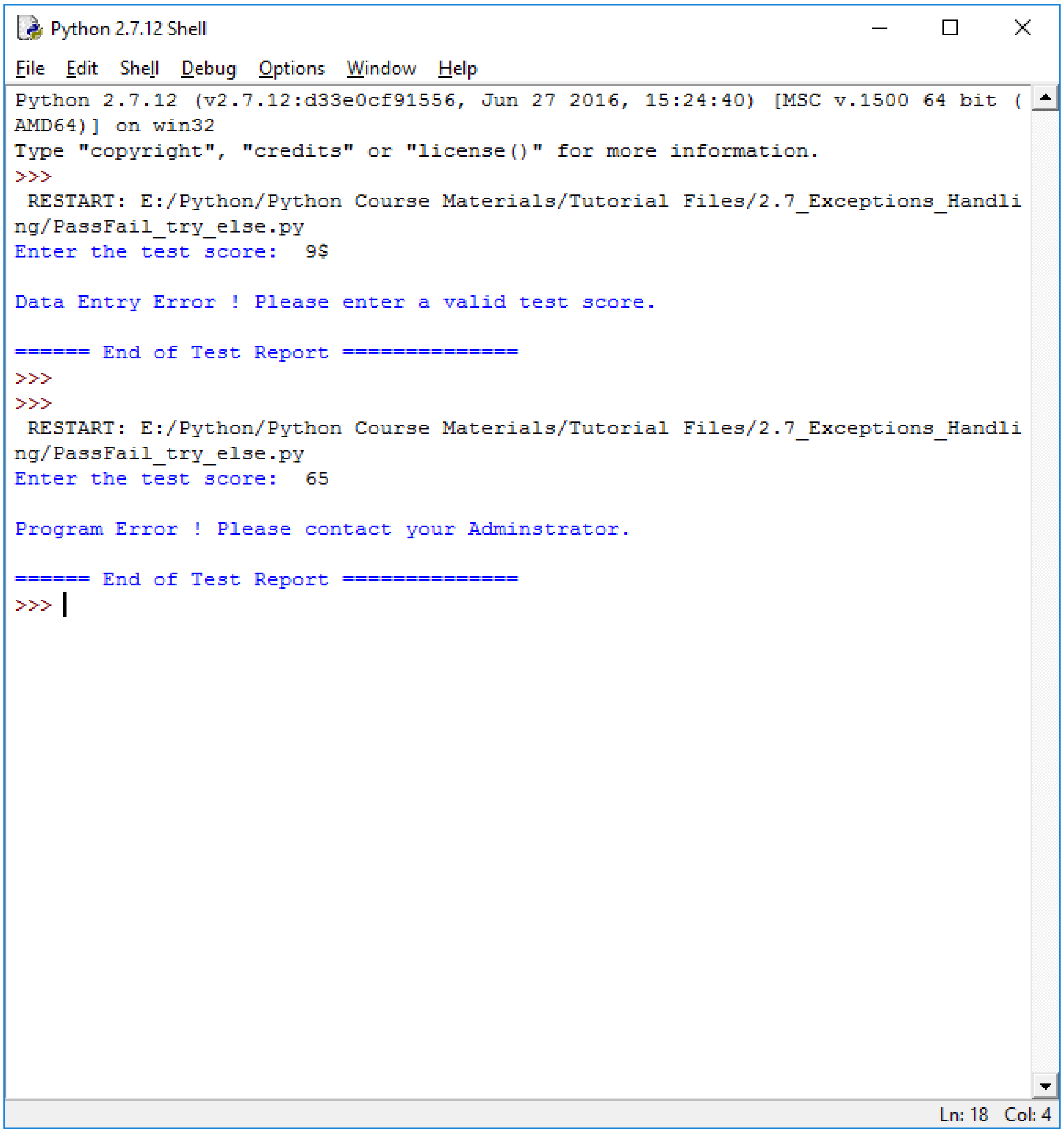
If the programmer anticipates multiple specific errors, they may be incorporated into the error handler as follows.


*PassFail_try_else.py - E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Hand...

```
#===========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#===========================================

try:
    # enter test score at input prompt
    x = float(input('Enter the test score:  '))

    # if statement to 'catch' a pass
    if x >= 70:
        grade = 'PASS'
        comment = 'Well Done.'

    # if statement to 'catch' a fail
    if y < 70:
        grade = 'FAIL'
        comment = 'Pleae retake the test'

    # print out results and comments
    print(grade)
    print(' ')
    print(comment)

except NameError:  # show the following if name error occurs
    print(' ')
    print('Program Error ! Please contact your Adminstrator.')

except SyntaxError:  # show the following if syntax error occurs
    print(' ')
    print('Data Entry Error ! Please enter a valid test score.')

else:  # show the following if no exception occurs
       # or if none of the above occur
    print(' ')
    print('Grade conversion processed.')

print(' ')
print('====== End of Test Report ==============')
```

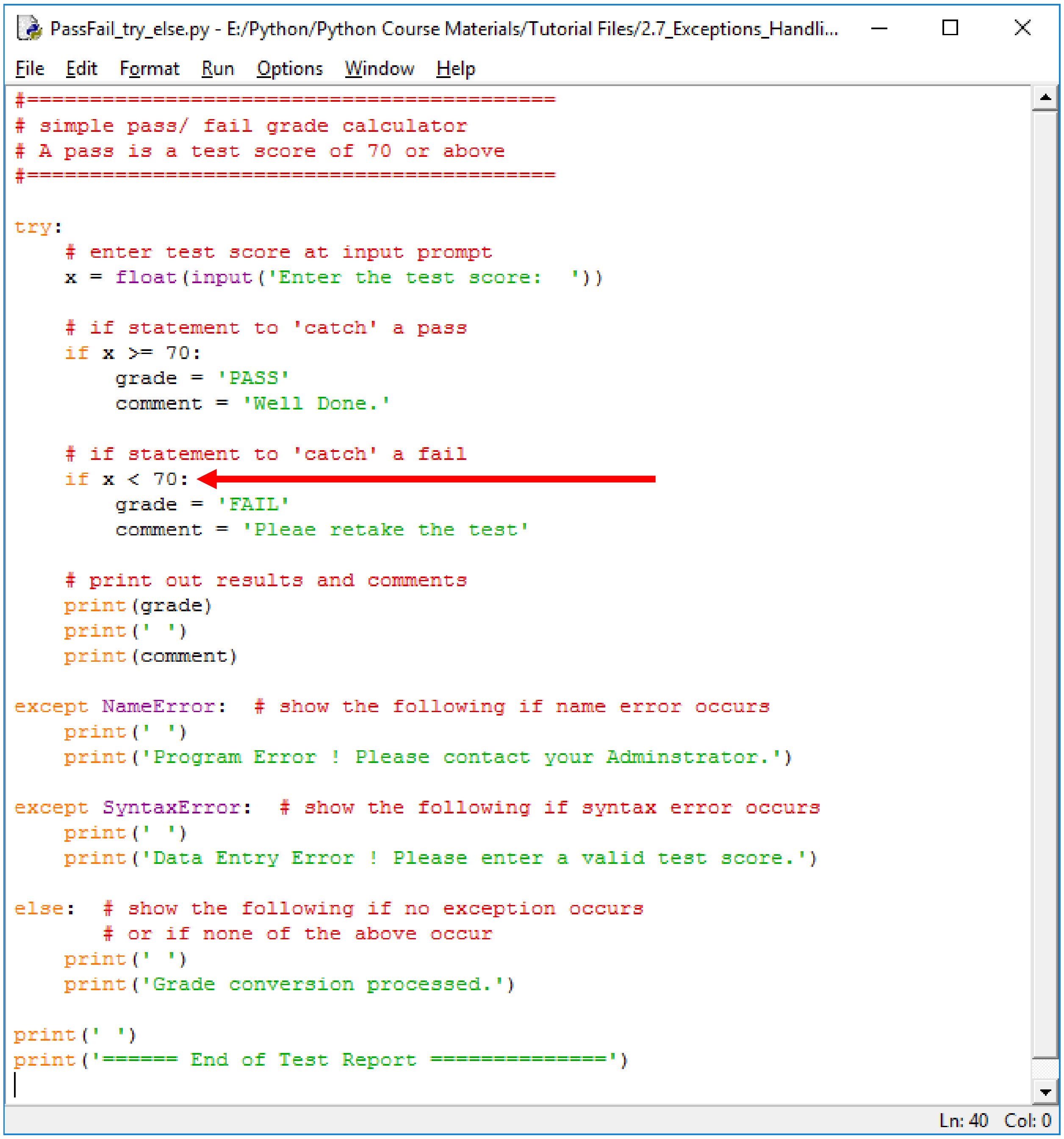
will raise a NameError, as no variable *y* exists

Run the file as follows.



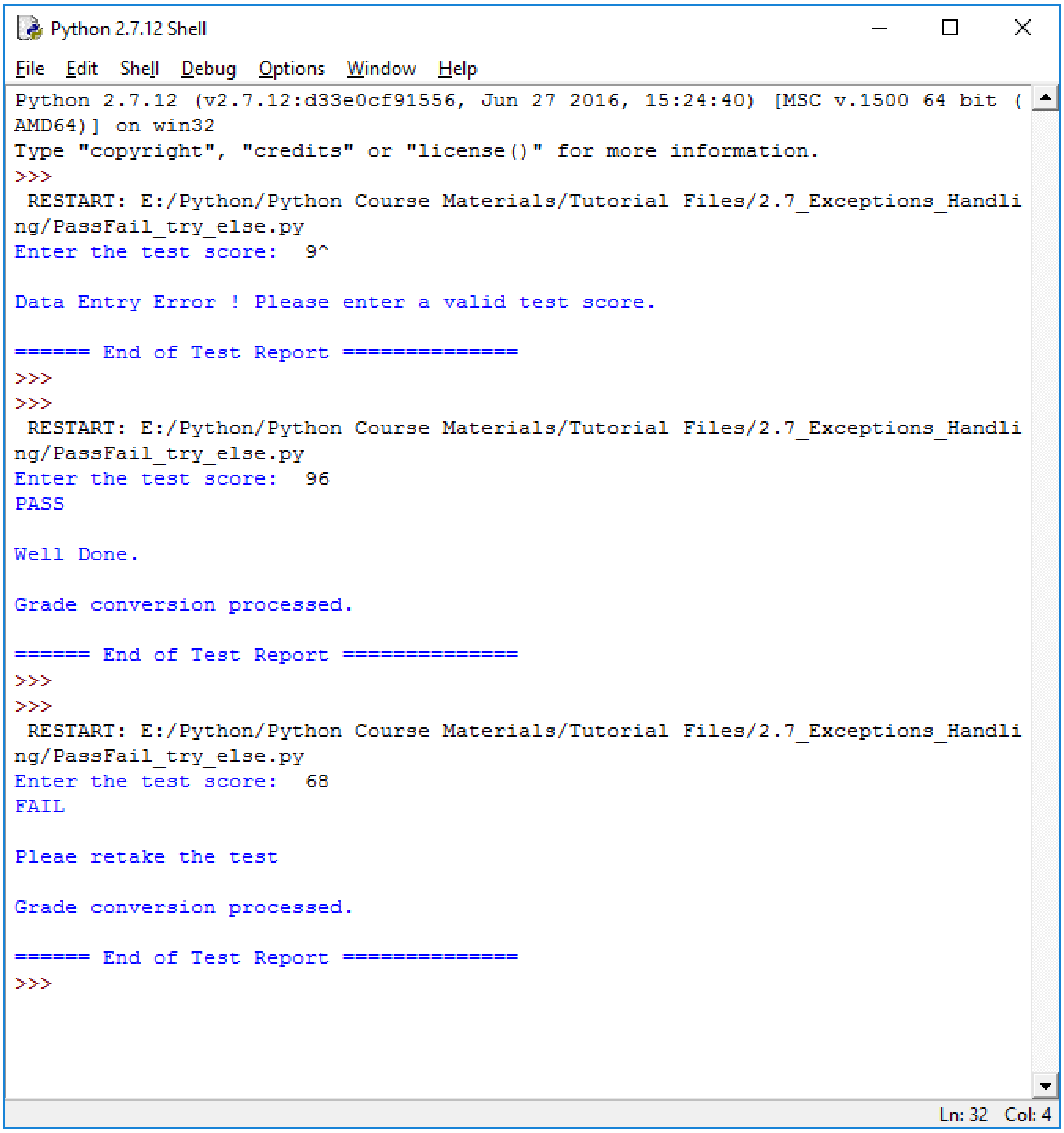
```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli
ng/PassFail_try_else.py 
Enter the test score:  9$

Data Entry Error ! Please enter a valid test score.

====== End of Test Report ==============
>>> 
>>> 
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli
ng/PassFail_try_else.py 
Enter the test score:  65

Program Error ! Please contact your Adminstrator.

====== End of Test Report ==============
>>> 
```

Fix the NameError issue.


PassFail_try_else.py - E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli...
File Edit Format Run Options Window Help
Ln: 40 Col: 0

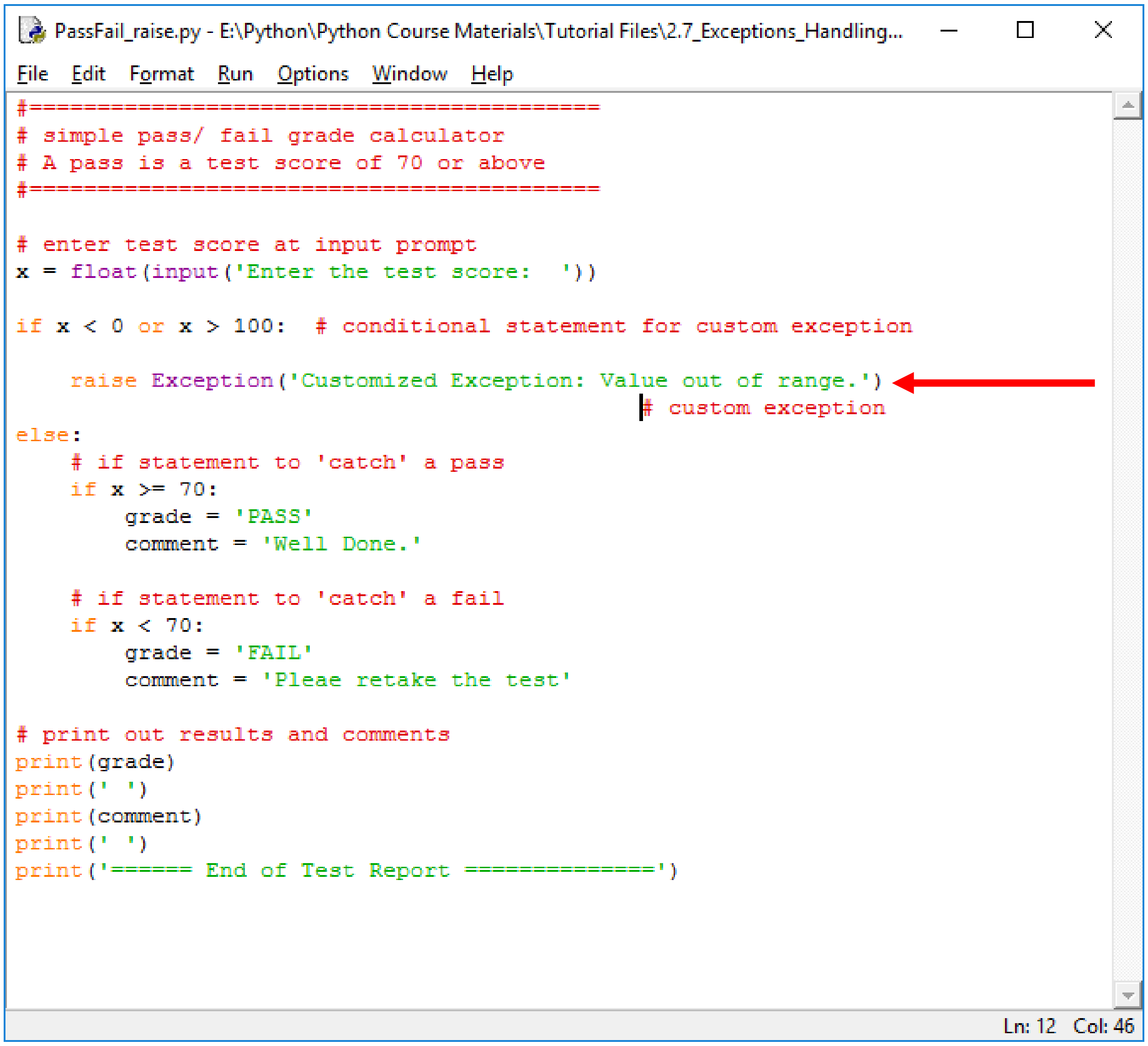
```
#==========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#==========================================

try:
    # enter test score at input prompt
    x = float(input('Enter the test score:  '))

    # if statement to 'catch' a pass
    if x >= 70:
        grade = 'PASS'
        comment = 'Well Done.'

    # if statement to 'catch' a fail
    if x < 70:
        grade = 'FAIL'
        comment = 'Pleae retake the test'

    # print out results and comments
    print(grade)
    print(' ')
    print(comment)

except NameError:  # show the following if name error occurs
    print(' ')
    print('Program Error ! Please contact your Adminstrator.')

except SyntaxError:  # show the following if syntax error occurs
    print(' ')
    print('Data Entry Error ! Please enter a valid test score.')

else:  # show the following if no exception occurs
       # or if none of the above occur
    print(' ')
    print('Grade conversion processed.')

print(' ')
print('====== End of Test Report ==============')
```

Run the file as follows.



```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli
ng/PassFail_try_else.py 
Enter the test score:  9^

Data Entry Error ! Please enter a valid test score.

====== End of Test Report ==============
>>> 
>>> 
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli
ng/PassFail_try_else.py 
Enter the test score:  96
PASS

Well Done.

Grade conversion processed.

====== End of Test Report ==============
>>> 
>>> 
 RESTART: E:/Python/Python Course Materials/Tutorial Files/2.7_Exceptions_Handli
ng/PassFail_try_else.py 
Enter the test score:  68
FAIL

Pleae retake the test

Grade conversion processed.

====== End of Test Report ==============
>>> 
```

Revert to your original PassFail app (with no error handlers).
Create a custom exception as follows.

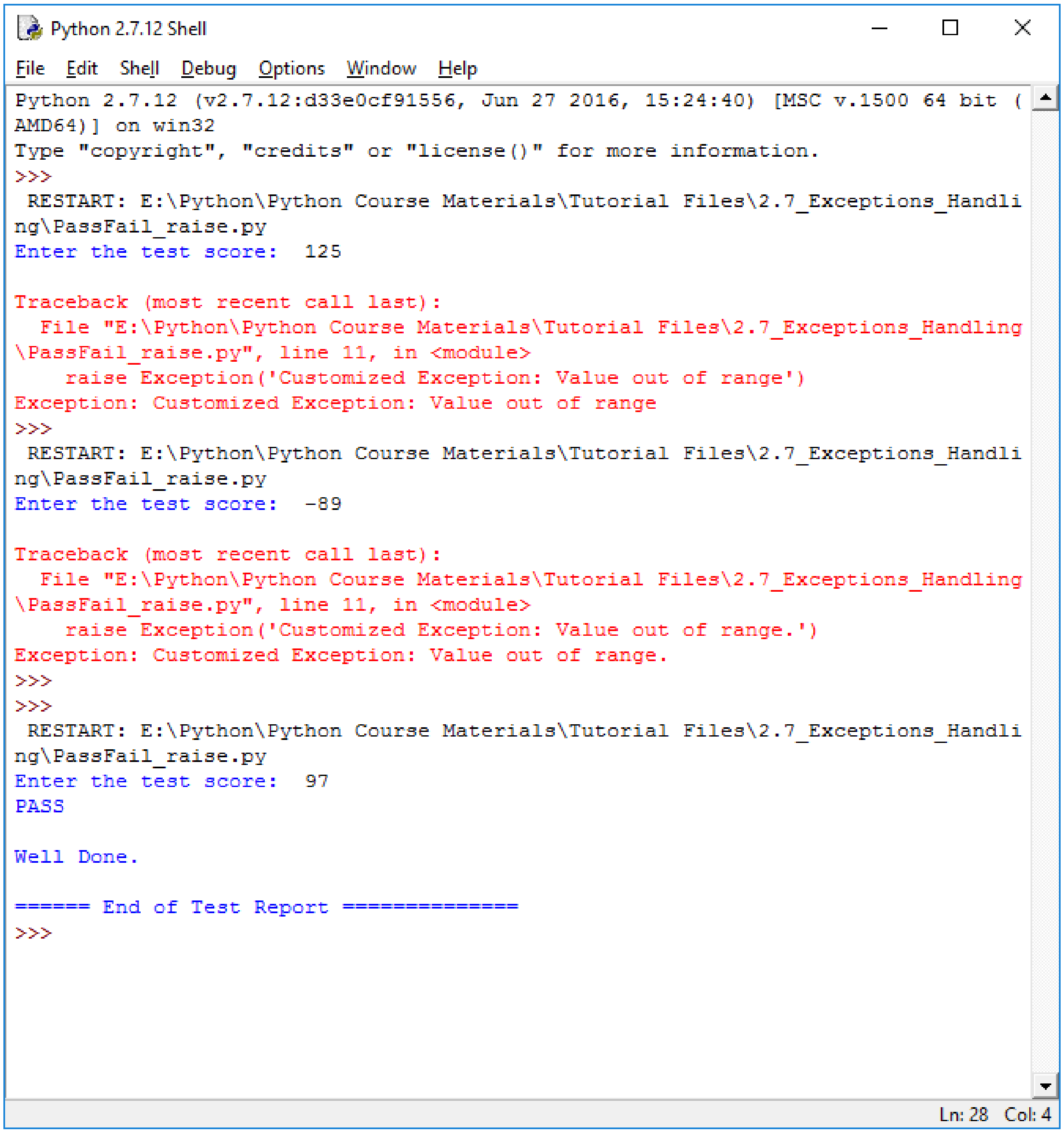


PassFail_raise.py - E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling...

```
#==========================================
# simple pass/ fail grade calculator
# A pass is a test score of 70 or above
#==========================================

# enter test score at input prompt
x = float(input('Enter the test score:  '))

if x < 0 or x > 100:  # conditional statement for custom exception

    raise Exception('Customized Exception: Value out of range.')
                                                  # custom exception
else:
    # if statement to 'catch' a pass
    if x >= 70:
        grade = 'PASS'
        comment = 'Well Done.'

    # if statement to 'catch' a fail
    if x < 70:
        grade = 'FAIL'
        comment = 'Pleae retake the test'

# print out results and comments
print(grade)
print(' ')
print(comment)
print(' ')
print('====== End of Test Report ==============')
```

Ln: 12   Col: 46

Run the file as follows.



```
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:24:40) [MSC v.1500 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail_raise.py 
Enter the test score:  125

Traceback (most recent call last):
  File "E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling
\PassFail_raise.py", line 11, in <module>
    raise Exception('Customized Exception: Value out of range')
Exception: Customized Exception: Value out of range
>>> 
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail_raise.py 
Enter the test score:  -89

Traceback (most recent call last):
  File "E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handling
\PassFail_raise.py", line 11, in <module>
    raise Exception('Customized Exception: Value out of range.')
Exception: Customized Exception: Value out of range.
>>> 
>>> 
 RESTART: E:\Python\Python Course Materials\Tutorial Files\2.7_Exceptions_Handli
ng\PassFail_raise.py 
Enter the test score:  97
PASS

Well Done.

====== End of Test Report ==============
>>> 
```

Domain knowledge and experience must be used to design and implement appropriate error handlers. The programmer should test multiple scenarios that a non-expert end-user may encounter and design the error handlers to guide and assist the end-user accordingly while precluding to the maximum extent possible, built-in exceptions being raised to the end-user.

8.9 Debugging *Python* Scripts

It cannot be overemphasized that all *Python* scripts should be meticulously reviewed, thoroughly scrutinized, and frequently tested as they are being developed. It is the programmer's responsibility to frequently test the code and address problems as they arise, and to verify or otherwise that the scripts execute as intended (validation). Test your codes and scripts frequently, block by block, line by line, using the IDLE (Python GUI) or the File Editor or any other preferred Python tool. A piecemeal approach to writing and testing code is strongly preferred rather than writing the entire script before testing it. In the latter scenario, it will be significantly more difficult to identify and isolate the relevant problems.

8.10 Getting Help

There is currently an abundance of help information on *Python* programming, particularly on the World Wide Web. These include official (peer-reviewed) and unofficial sources, websites, academic reports, professional presentations, tutorial videos (YouTube, etc.), user groups, online forums, downloadable code snippets, etc., etc. Typing a *Python* topic in a search engine will typically yield hundreds if not thousands of results. It is still strongly recommended, regardless of the source of any contributory or relevant help information, that all codes being developed be tested and validated thoroughly before deployment.

9. CONCLUSION

This course has presented a broad overview of fundamental concepts, principles, and programming structures of the *Python* programming language. *Python* is an interpreted, high-level, general purpose programming language. *Python* is a free and open source and can be used to build a wide range of desktop and web-based applications.

In this course the topics the following topics were presented in detail: conditional statements, looping, functions, input and output (I/O) functions, modules, file handling and error handling techniques. Practical examples from situations encountered by a practicing engineer or scientist were used to illustrate and demonstrate the concepts and methods learned in this class.

This course has prepared participants to now develop their own applications driven by *Python*. This course has enabled participants to identify situations where computer programming is relevant and will be of advantage to the practicing professional competing in the global marketplace.

Practitioners are strongly encouraged to look for situations in their domains of expertise where computer programming solutions are applicable and will be of benefit to their work and their organizations.

All programming requires a careful and meticulous approach and can only be mastered and retained by practice and repetition.

Good Luck and Happy Programming.

REFERENCES

Programiz. (2019). *Python Errors and Built-in Exceptions*. Retrieved July 2019, from Programiz: https://www.programiz.com/python-programming/exceptions

Python Software Foundation. (2019). *Python Software Foundation*. Retrieved July 2019, from Python Software Foundation: http://www.python.org/psf/

Real Python. (2019). *Python Tutorials - Real Python*. Retrieved August 2019, from Real Python: http://realpython.com/

tutorialspoint. (2019). *Python - Tutorial*. Retrieved June 2019, from tutorialspoint.com: https://www.tutorialspoint.com/python/

w3schools.com. (2019). *Python Tutorial*. Retrieved July 2019, from w3schools.com: http://www.w3schools.com/python/default.asp

Images were all drawn/prepared by Kwabena. Ofosu