**Project1**

## Part 1

**Development:**

1. Requirements:
   a. Code and run **demo.py. (Note: *be sure* necessary packages are installed!)**
   **b. Use it to backward-engineer the screenshots below it.**
   c. Update **conda**, and install necessary Python packages. Invoke commands below.
   Log in as administrator (**PC: Anaconda** command prompt, **LINUX/Mac:** sudo)
      1) `conda update conda`
      2) `conda update --all`
      3) `pip install pandas-datareader`

2. Be sure to test your program using both **IDLE** and **Visual Studio Code**.

## Part 2

**README.md** file should include the following items:

1. **Assignment requirements, as per A1.**
2. Screenshots as per example below, **including graph**.
3. Upload P1 **.ipynb** file and create link in README.md;
   **Note:** *Before* uploading **.ipynb** file, ***be sure*** to do the following actions from **Kernal** menu:
      a. **Restart & Clear Output**
      b. **Restart & Run All**

**Deliverables:**

1. Provide **Bitbucket** read-only access to **lis4369** repo, include links to the repos you created in the above tutorials in **README.md**, using Markdown syntax
   (**README.md** must also include screenshots as per above.)
2. **FSU's Learning Management System**: lis4369 **Bitbucket** repo

**demo.py**

```
# Pandas = "Python Data Analysis Library"
# Be sure to: pip install pandas-datareader
import datetime
import pandas_datareader as pdr  # remote data access for pandas
import matplotlib.pyplot as plt
from matplotlib import style

start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2018, 10, 15)
# for "end": *must* use Python function for current day/time

# Read data into Pandas DataFrame
# NOTE: XOM is stock market symbol for Exxon Mobil Corporation
df = pdr.DataReader("XOM", "yahoo", start, end)

print("\nPrint number of records: ")
# statement goes here...

# Why is it important to run the following print statement...
print(df.columns)

print("\nPrint data frame: ")
print(df)  # Note: for efficiency, only prints 60--not *all* records

print("\nPrint first five lines:")
# Note: "Date" is lower than the other columns as it is treated as an index
# statement goes here...

print("\nPrint last five lines:")
# statement goes here...

print("\nPrint first 2 lines:")
# statement goes here...

print("\nPrint last 2 lines:")
# statement goes here...

# Research what these styles do!
# style.use('fivethirtyeight')
# compare with...
style.use('ggplot')

df['High'].plot()
df['Adj Close'].plot()
plt.legend()
plt.show()
```

**Assignment Requirements**

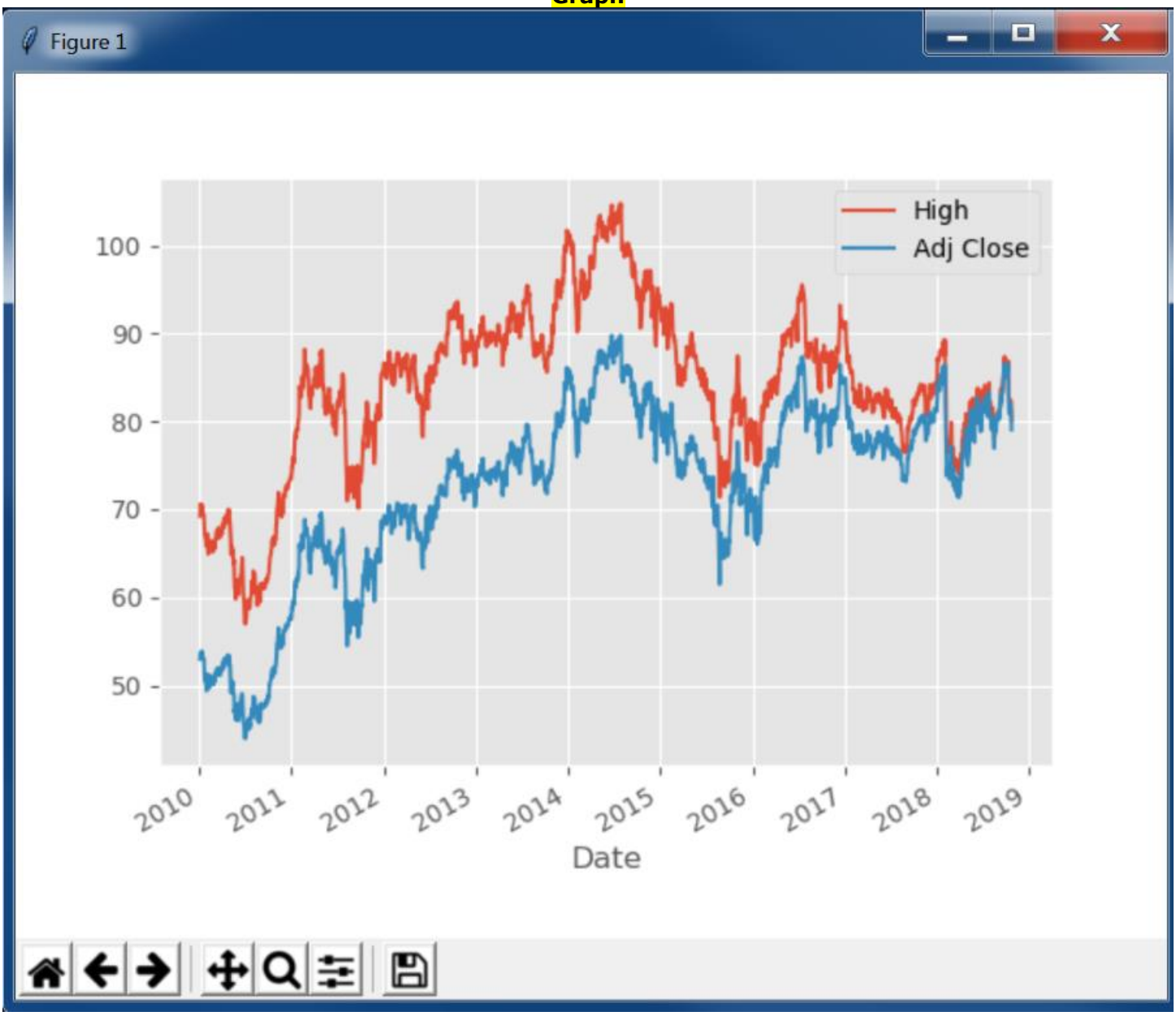
(***Be sure*** to include *your* name as "Developer"!)

```
Data Analysis 1

Program Requirements:
1. Run demo.py.
2. If errors, more than likely missing installations.
3. Test Python Package Installer: pip freeze
4. Research how to do the following installations:
        a. pandas (only if missing)
        b. pandas-datareader (only if missing)
        c. matplotlib (only if missing)
5. Create at least three functions that are called by the program:
        a. main(): calls at least two other functions.
        a. get_requirements(): displays the program requirements.
        c. data_analysis_1(): displays the following data.

Print number of records:
2218

Print columns:
Index(['High', 'Low', 'Open', 'Close', 'Volume', 'Adj Close'], dtype='object')

Print data frame:
                High        Low       Open      Close      Volume  Adj Close
Date
2010-01-04  69.260002  68.190002  68.720001  69.150002  27809100.0  52.979797
2010-01-05  69.449997  68.800003  69.190002  69.419998  30174700.0  53.186668
2010-01-06  70.599998  69.339996  69.449997  70.019997  35044700.0  53.646351
2010-01-07  70.059998  69.419998  69.900002  69.800003  27192100.0  53.477802
2010-01-08  69.750000  69.220001  69.690002  69.519997  24891800.0  53.263279
2010-01-11  70.519997  69.650002  69.940002  70.300003  30685000.0  53.860882
2010-01-12  69.989998  69.519997  69.720001  69.949997  31496700.0  53.592728
2010-01-13  70.040001  69.260002  69.959999  69.669998  24884400.0  53.378204
2010-01-14  69.739998  69.349998  69.540001  69.680000  18630800.0  53.385845
2010-01-15  69.690002  68.650002  69.650002  69.110001  29411900.0  52.949154
2010-01-19  69.349998  68.419998  68.739998  69.269997  26081900.0  53.071732
2010-01-20  68.660004  67.930000  68.559998  68.029999  34629500.0  52.121693
2010-01-21  68.150002  66.500000  68.120003  66.699997  39114000.0  51.102715
2010-01-22  67.139999  66.000000  66.519997  66.099998  39085500.0  50.643032
2010-01-25  66.760002  65.690002  66.550003  65.849998  29305100.0  50.451485
2010-01-26  66.739998  65.500000  65.639999  65.919998  34083300.0  50.505108
2010-01-27  66.059998  65.000000  65.660004  65.540001  35723500.0  50.213970
2010-01-28  65.849998  64.570000  65.849998  64.959999  37349800.0  49.769604
2010-01-29  65.820000  64.019997  65.150002  64.430000  40880500.0  49.363537
2010-02-01  66.410004  65.349998  65.769997  66.180000  37703000.0  50.704319
2010-02-02  67.120003  66.470001  66.730003  66.959999  34057900.0  51.301918
2010-02-03  67.230003  66.559998  66.879997  66.599998  24024700.0  51.026096
2010-02-04  66.349998  64.680000  66.269997  64.720001  33858200.0  49.585724
2010-02-05  64.900002  63.560001  64.699997  64.800003  42297500.0  49.647018
2010-02-08  65.489998  64.339996  64.910004  64.349998  30519400.0  49.623878
2010-02-09  65.709999  64.559998  65.110001  65.199997  36243300.0  50.279358
2010-02-10  65.220001  64.160004  65.080002  64.849998  21699100.0  50.009468
2010-02-11  65.480003  64.410004  64.690002  65.239998  23555200.0  50.310200
2010-02-12  65.139999  64.279999  64.620003  64.800003  30636200.0  49.970905
2010-02-16  66.379997  65.080002  65.440002  66.279999  30514900.0  51.112221
```

```
...              ...        ...        ...        ...         ...        ...
2018-09-12  83.779999  82.870003  83.250000  83.129997  11556300.0  83.129997
2018-09-13  83.250000  82.010002  83.150002  82.320000  11162700.0  82.320000
2018-09-14  83.150002  82.269997  82.419998  82.919998   9442600.0  82.919998
2018-09-17  83.610001  82.989998  83.000000  83.410004   8403300.0  83.410004
2018-09-18  84.129997  83.449997  83.900002  83.629997   9219900.0  83.629997
2018-09-19  84.769997  83.639999  83.639999  84.629997  10413600.0  84.629997
2018-09-20  85.339996  84.540001  84.860001  84.820000  11198200.0  84.820000
2018-09-21  85.430000  84.519997  85.010002  85.169998  26639400.0  85.169998
2018-09-24  87.089996  85.720001  85.790001  86.599998  13549500.0  86.599998
2018-09-25  87.360001  86.370003  87.029999  86.500000  12200700.0  86.500000
2018-09-26  86.500000  85.690002  86.019997  85.779999  10275500.0  85.779999
2018-09-27  86.379997  85.589996  86.089996  85.769997   7895400.0  85.769997
2018-09-28  85.930000  84.989998  85.250000  85.019997   9884900.0  85.019997
2018-10-01  86.029999  85.260002  85.349998  85.809998   8566900.0  85.809998
2018-10-02  86.669998  85.620003  85.800003  86.459999   8453100.0  86.459999
2018-10-03  86.889999  85.980003  86.510002  86.150002  10206700.0  86.150002
2018-10-04  86.080002  85.250000  85.500000  85.580002  10204600.0  85.580002
2018-10-05  85.699997  84.930000  85.309998  85.339996   9217400.0  85.339996
2018-10-08  86.309998  84.650002  84.790001  86.129997  13242400.0  86.129997
2018-10-09  86.879997  85.739998  86.379997  86.510002  10177900.0  86.510002
2018-10-10  86.820000  84.500000  86.730003  84.519997  16573400.0  84.519997
2018-10-11  84.169998  81.169998  83.940002  81.599998  20320100.0  81.599998
2018-10-12  82.239998  80.269997  82.129997  81.379997  15216300.0  81.379997
2018-10-15  81.739998  80.820000  81.379997  80.820000  10558800.0  80.820000
2018-10-16  81.269997  80.010002  80.510002  81.199997   9781800.0  81.199997
2018-10-17  81.519997  80.339996  80.940002  81.500000  12247700.0  81.500000
2018-10-18  82.470001  81.199997  81.199997  81.849998  17448600.0  81.849998
2018-10-19  82.459999  81.510002  81.660004  81.970001  12098700.0  81.970001
2018-10-22  82.180000  80.639999  82.000000  81.150002   8758100.0  81.150002
2018-10-23  80.120003  78.719902  80.059998  79.040001   5377696.0  79.040001

[2218 rows x 6 columns]

Print first five lines:
                High        Low       Open      Close      Volume  Adj Close
Date
2010-01-04  69.260002  68.190002  68.720001  69.150002  27809100.0  52.979797
2010-01-05  69.449997  68.800003  69.190002  69.419998  30174700.0  53.186668
2010-01-06  70.599998  69.339996  69.449997  70.019997  35044700.0  53.646351
2010-01-07  70.059998  69.419998  69.900002  69.800003  27192100.0  53.477802
2010-01-08  69.750000  69.220001  69.690002  69.519997  24891800.0  53.263279

Print last five lines:
                High        Low       Open      Close      Volume  Adj Close
Date
2018-10-17  81.519997  80.339996  80.940002  81.500000  12247700.0  81.500000
2018-10-18  82.470001  81.199997  81.199997  81.849998  17448600.0  81.849998
2018-10-19  82.459999  81.510002  81.660004  81.970001  12098700.0  81.970001
2018-10-22  82.180000  80.639999  82.000000  81.150002   8758100.0  81.150002
2018-10-23  80.120003  78.719902  80.059998  79.040001   5377696.0  79.040001
```

```
Print first 2 lines:
                High        Low       Open      Close      Volume  Adj Close
Date
2010-01-04  69.260002  68.190002  68.720001  69.150002  27809100.0  52.979797
2010-01-05  69.449997  68.800003  69.190002  69.419998  30174700.0  53.186668

Print last 2 lines:
                High        Low       Open      Close     Volume  Adj Close
Date
2018-10-22  82.180000  80.639999  82.000000  81.150002  8758100.0  81.150002
2018-10-23  80.120003  78.719902  80.059998  79.040001  5377696.0  79.040001
```

**Graph**



## Part 3
## Questions (Python: Chs. 7, 8):

1. A binary file is like a text file in all but one of the following ways. Which one is it?
   A binary file stores numbers with binary notation.
   A binary file stores strings with character notation.
   The data in a binary file can be grouped into records or rows.
   A binary file can be used to store a Python list.

2. Consider the following code:

```
import csv
def main():
    courses = [["Python", 3],
               ["Trig", 3],
               ["Physics", 4],
               ["Yoga", 2]]
    with open("courses.csv", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerows(courses)
    course_list = []
    with open("courses.csv", newline="") as file:
        reader = csv.reader(file)
        for row in reader:
            course_list.append(row)
    for i in range(len(course_list) - 2):
        course = course_list[i]
        print(course[0] + " (" + str(course[1]) + ")")

main()
```

What happens if the courses.csv file doesn't exist when the first with open statement is executed?
   the program crashes
   an exception is thrown but the program doesn't crash
   a new file named courses.csv is created
   all the resources for the file are released

3. Consider the following code:

```
import csv
def main():
    courses = [["Python", 3],
               ["Trig", 3],
               ["Physics", 4],
               ["Yoga", 2]]
    with open("courses.csv", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerows(courses)
    course_list = []
    with open("courses.csv", newline="") as file:
        reader = csv.reader(file)
        for row in reader:
            course_list.append(row)
    for i in range(len(course_list) - 2):
        course = course_list[i]
        print(course[0] + " (" + str(course[1]) + ")")

main()
```

If the first with open statement works, what is written to the file?

The list named courses.
The first list in the list named courses.
The first row in the list named courses.
The first column in the first row in the list named courses.

4. Consider the following code:

```
import csv
def main():
    courses = [["Python", 3],
               ["Trig", 3],
               ["Physics", 4],
               ["Yoga", 2]]
    with open("courses.csv", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerows(courses)
    course_list = []
    with open("courses.csv", newline="") as file:
        reader = csv.reader(file)
        for row in reader:
            course_list.append(row)
    for i in range(len(course_list) - 2):
        course = course_list[i]
        print(course[0] + " (" + str(course[1]) + ")")

main()
```

What will display on the console after the code executes?

Python 3
Trig 3
Physics 4
Yoga 2

Python (3)
Trig (3)
Physics (4)
Yoga (2)

Python 3
Trig 3

Python (3)
Trig (3)

5. Consider the following code:

```
import pickle
def main():
   courses = [["Python", 3],
              ["Trig", 3],
              ["Physics", 4],
              ["Yoga", 2]]
   with open("classes.bin", "wb") as file:
      pickle.dump(courses, file)
   with open("classes.bin", "rb") as file:
      course_list = pickle.load(file)
   i = 0
   while i < len(course_list):
      course = course_list[i]
      print(course[0], str(course[1]), end=" ")
      i += 2

main()
```

What does the first with open statement do?

- writes the courses list to a binary file if the file named classes.bin doesn't exist
- causes an exception if the file named classes.bin doesn't exist
- writes the courses list to a binary file if the file named courses.bin doesn't exist
- causes an exception if the file named courses.bin doesn't exist

6. Consider the following code:

```
import pickle
def main():
   courses = [["Python", 3],
              ["Trig", 3],
              ["Physics", 4],
              ["Yoga", 2]]
   with open("classes.bin", "wb") as file:
      pickle.dump(courses, file)
   with open("classes.bin", "rb") as file:
      course_list = pickle.load(file)
   i = 0
   while i < len(course_list):
      course = course_list[i]
      print(course[0], str(course[1]), end=" ")
      i += 2

main()
```

What does the second with open statement do?

- reads the file named classes.bin into the list named courses
- causes an exception if the file named classes.bin doesn't exist
- reads the list named courses into the list named course_list
- creates an empty list if the file named classes_bin doesn't exist

7. Consider the following code:

```
import pickle
def main():
    courses = [["Python", 3],
               ["Trig", 3],
               ["Physics", 4],
               ["Yoga", 2]]
    with open("classes.bin", "wb") as file:
        pickle.dump(courses, file)
    with open("classes.bin", "rb") as file:
        course_list = pickle.load(file)
    i = 0
    while i < len(course_list):
        course = course_list[i]
        print(course[0], str(course[1]), end=" ")
        i += 2

main()
```

What is displayed on the console by the while loop?

Python 3
Trig 3
Physics 4
Yoga 2

Python 3 Trig 3 Physics 4 Yoga 2

Python 3
Physics 4

Python 3 Physics 4

8. Given the following 2-dimensional list of 3 rows and 3 columns, how would you write this list to a CSV file named prog.csv?

```
programming = [["Python", "cop1000", 3],
               ["Java", "cop1020", 3],
               ["HTML5", "cop1040", 3]]
    open("prog.csv", "w", newline ="") as csv_file:
    writer = csv.writer(file)
    writer.writerows(programming)
     with open("programming.csv", "w", newline ="") as programming:
    writer = csv.writer(prog.csv)
    writer.writerows(programming)
     with open("prog.csv", "w", newline ="") as file:
    writer = csv.writer(file)
    writer.writerows(programming)
     with open("prog.csv", "w", newline ="") as programming:
    writer = csv.writer(programming)
    writer.writerows(prog.csv)
```

9. To read a list of lists that's stored in a binary file, you use

the load() method of the binary module
the read() method of the binary module
the load() method of the pickle module
the read() method of the pickle module

10. To read the rows in a CSV file, you need to
    get a reader object by using the reader() function of the file object
    get a reader object by using the reader() function of the csv module
    get a row object by using the row() function of the file object
    get a rows object by using the rows() function of the file object

11. To work with a file when you're using Python, you must do all but one of the following. Which one is it?
    open the file
    write data to or read data from the file
    decode the data in the file
    close the file

12. Which of the following is not true about a CSV file?
    Each row or record in the file usually ends with a new line character.
    The columns or fields are usually separated by commas.
    The csv module is a standard module so you don't need to import it
    To write data to a file, you need to get a writer object.

13. Which one of the following is not a benefit of using a with statement to open a file?
    The file is automatically closed.
    This file is closed even if an exception occurs while the file is being processed.
    The resources used by the file are released when the file is closed.
    You don't have to specify the path for the file.

14. A Python program should use try statements to handle
    all exceptions that might be thrown by a program
    only the exceptions related to file and database I/O
    all the exceptions that aren't caused by coding errors
    all exceptions that can't be prevented by normal coding techniques

15. Consider the following code:

```
import csv
import sys
FILENAME = "names.csv"
def main():
   try:
      names = []
      with open(FILENAME, newline="") as file:
         reader = csv.reader(file)
         for row in reader:
            names.append(row)
   except FileNotFoundError as e:
      print("Could not find " + FILENAME + " file.")
      sys.exit()
   except Exception as e:
      print(type(e), e)
      sys.exit()
   print(names)
if __name__ == "__main__":
   main()
```

If the names.csv file is not in the same directory as the file that contains the Python code, what type of exception will be thrown and caught?
    Exception
    OSError
    FileNotFoundError
    All of the above

16. Consider the following code:

```
import csv
import sys
FILENAME = "names.csv"
def main():
    try:
        names = []
        with open(FILENAME, newline="") as file:
            reader = csv.reader(file)
            for row in reader:
                names.append(row)
    except FileNotFoundError as e:
        print("Could not find " + FILENAME + " file.")
        sys.exit()
    except Exception as e:
        print(type(e), e)
        sys.exit()
    print(names)
if __name__ == "__main__":
    main()
```

If the for statement in the try clause refers to readers instead of reader, what type exception will be thrown and caught?

- Exception
- SyntaxError
- NameError
- ReferenceError

17. If a program attempts to read from a file that does not exist, which of the following will catch that error?

- FileNotFoundError and ValueError
- FileNotFoundError and NameError
- FileNotFoundError and OSError
- NameError and OSError

18. It's a common practice to throw your own exceptions to test error handling routines that

- provide for many different types of exceptions
- catch exceptions that are hard to produce otherwise
- handle many varieties of input
- that handle complexities like lists within lists

19. The finally clause of a try statement

- is required
- is executed whether or not an exception has been thrown
- can be used to display more information about an exception
- can be used to recover from an exception

20. To cancel the execution of a program in the catch clause of a try statement, you can use the

- cancel() function of the sys module
- exit() function of the sys module
- the built-in cancel() function
- the built-in exit() function

21. To throw an exception with Python code, you use the
    throw statement
    raise statement
    built-in throw() function
    build-in raise() function

22. When an exception is thrown, Python creates an exception object that contains all but one of the following items of information. Which one is it?
    the type of exception
    the name of the class for the type of exception
    the severity of the exception
    the message for the exception

23. Which of the following is the correct way to code a try statement that catches any type of exception that can occur in the try clause?

```
    try:
   number = float(input("Enter a number: "))
   print("Your number is: ", number)
except:
    print("Invalid number.")
    try:
   number = float(input("Enter a number: "))
   print("Your number is: ", number)
except ValueError:
    print("Invalid number.")
    try:
   number = float(input("Enter a number: "))
   print("Your number is: ", number)
    try:
   number = float(input("Enter a number: "))
   print("Your number is: ", number)
else:
    print("Invalid number.")
```

24. Which of the following is the correct way to code a try statement that displays the type and message of the exception that's caught?

```
    try:
   number = int(input("Enter a number: "))
   print("Your number is: ", number)
except Exception as e:
   print(e(type), e(message))
    try:
   number = int(input("Enter a number: "))
   print("Your number is: ", number)
except Exception as e:
   print(type(e), e)
    try:
   number = int(input("Enter a number: "))
   print("Your number is: ", number)
except Exception:
    print(Exception(type), Exception(message))
    try:
   number = int(input("Enter a number: "))
   print("Your number is: ", number)
except Exception:
    print(type(Exception), Exception)
```

25. Within the try clause of a try statement, you code
    all the statements of the program
    only the statements that might cause an exception
    a block of statements that might cause an exception
    a block of the statements that are most likely to cause an exception