Python Programming

Homework 3

Instructor: Dr. Ionut Cardei

The following problems are from Chapters 6, 7, and 8.

For full credit:

- make sure you follow exactly all requirements listed in this document,
- the program must have all required features and run as expected.

Write your answers into a new Word document called **h3.doc**.
Write your full name at the top of the file and add a heading before each problem solution.

Don't forget to read the **Important Notes** at the end of this file.

Problem 1. Tuple Input

a) Write a function called *input_tuple (*in a new file **p1.py)** that reads from the terminal a sequence of objects with types provided by a tuple given as parameter and that returns the sequence of objects read as a tuple.
The function will iterate through the types tuple and will try to parse the input string based on the types.

The function takes the following parameters:

- *prompt*: a string to be displayed as input prompt
- *types*: a tuple of type objects that must match the types of the data entered by the user and returned by the function. The only types supported are *int, float, str, bool*
- *sep*: a string that separates the input objects, with default value ",".

The function returns one of:

- if parsing the data from the user according to the *types* tuple **succeeds**, then it returns the tuple with the converted data to Python objects,
- if parsing the data from the user according to the *types* tuple **fails**, then it returns the empty tuple **( )**.

Error handling: the function must not raise/throw errors related to parsing or input/read operations. Use proper error handling with try/except. In case of any error, print an error message to the terminal and return the empty tuple ().

Here is a successful example on how the function can be used:

```
student_tup = input_tuple("Enter first name, last name, age (float), ID (int), fulltime (bool): ",
                    (str, str, float, int, bool),     # this is the tuple with expected types
                    “,”)                               # , is the separator character on the input line
```

The function displays the prompt:
“Enter first name, last name, age (float), ID (int), fulltime (bool): “
The user enters the string: “Spongebob,Squarepants,23.5,1,True”. Then, the tuple returned will be equal to (“Spongebob”, “Squarepants”, 23.5, 1, True).

Here is an example when parsing based on the types tuple fails:
The function call is the same as above, but the user enters: “Spongebob Squarepants,23.5,1,True”. Notice that the token count is less than required – a separator is missing. The function returns ().
Another example of failure is if the user types: “Keith,Rupert,Murdoch,20,10010,False”. The function expects a float for the 3rd object, but gets instead a string that can’t be converted to float. The function returns (), too.

Function *input_tuple* should NOT use list comprehensions.

Hint: Python types (like *int, str*) can be used as constructors to ‘construct’ an object of a corresponding type from a string. For example, *int*(“30”) returns object of type int with number 30 inside.

b) Write a function called *input_tuple_lc* that is identical to *input_tuple* except that it uses list comprehension(s).

c) Write a function *read_tuple* that works similarly to *input_tuple*, but instead of reading input from the terminal, it reads text from a file object passed as argument. If this function uses correctly a list comprehension you get 2 extra points.

The function *read_tuple* takes the following parameters:

- *file_obj*: an object representing an open text file; e.g. opened with open(“filename”,”r”).
- *types*: a tuple of type objects that must match the types of the data entered by the user and returned by the function. The only types supported are *int, float, str, bool*
- *sep*: a string that separates the input objects, with default value “,”.

The function returns one of:

- if parsing the data from the user according to the *types* tuple **succeeds**, then it returns the tuple with the converted data
- if parsing the data from the user according to the *types* tuple **fails**, then it returns the empty tuple **( )**.

Error handling: the function must not raise/throw errors related to parsing or input/read operations. Use proper error handling with try/except. In case of any error, print an error message to the terminal and return the empty tuple ().

A usage scenario (omitting error handling for brevity) would be:

```
f = open(“cars.csv”, “r”)
(make, model, mpg_float, modelYr_int, newcar_bool) = read_tuple(f, (str, str, float, int, bool), “,”)
```

```
# use these variables ….
f.close()
```

This code will work fine if the file cars.csv looks like this:
Lada,Niva,19.5,1987,False
Porsche,911 Turbo S,17.5,1989,False

Function `read_tuple()` should return in this case the tuple ('Lada','Niva',19.5,1987,False) for the first time it's called.
If the input format does not comply with types argument `(str, str, float, int, bool)` then the function should return ().

c) In the main program (p1.py) write code that tests both functions.
Using the ***testif*** function from the Unit 2 module for writing your tests gives you **2 extra credit points**.

Hints:

- notice that MS Word uses a special type of the double quote " characters, in case you want to paste to a Python editor
- you can iterate with a *for* loop through the *types* tuple
- if a variable *t* holds a type object (from iterating through the types tuple, for instance), then you can convert a string s to an object of type *t* by calling *t*(*s*). If the conversion fails, it will throw/raise a ***ValueError***.
- *bool*(x) will succeed in most cases for different types of data. If x==0 or "", *bool*(*x*) will return *False*.
- a tuple is immutable, but a list is not. Grow a list first, then convert to a tuple.

Submission Instructions

- Write a heading for Problem 1 in file h3.doc.
- Insert the p1.py file in h3.doc.

Problem 2. Pythagorean Numbers Revisited

Write a function named *compute_Pythagoreans* in file p2.py that takes one positive *int* argument *n* and returns a list with tuples (*a*,*b*,*c*), with $0<a,b,c<=n$, such that $a^2+b^2=c^2$. The function MUST use **one list comprehension and no loops,** or no credit is given for the solution.

The function should also use proper error checking and return an empty list if the input parameter is invalid.

Submission Instructions

- Write a heading for Problem 2 in file h3.doc.
- Insert the p2.py file in h3.doc.

Problem 3. LeBron Worship

For this problem we will marvel at how glorious a basketball player LeBron James is.

Go to http://www.basketball-reference.com/players/j/jamesle01.html, and click on the “Share & More” menu, then click on the “Get table as CSV (for Excel)” menu item. Copy-paste-save the comma-separated-values table from the web page to a file to be named *lb-james.csv*.

The file should start with these 3 lines:

```
Season,Age,Tm,Lg,Pos,G,GS,MP,FG,FGA,FG%,3P,3PA,3P%,2P,2PA,2P%,eFG%,FT,FTA,FT%,ORB,DRB,TRB,AST,STL,BLK,TOV,PF,PTS
2003-04,19,CLE,NBA,SG,79,79,39.5,7.9,18.9,.417,0.8,2.7,.290,7.1,16.1,.438,.438,4.4,5.8,.754,1.3,4.2,5.5,5.9,1.6,0.7,3.5,1.9,20.9
2004-05,20,CLE,NBA,SF,80,80,42.4,9.9,21.1,.472,1.4,3.9,.351,8.6,17.2,.499,.504,6.0,8.0,.750,1.4,6.0,7.4,7.2,2.2,0.7,3.3,1.8,27.2
```

In case you wonder what the stats mean, the web page gives an explanation for each column. Just hover the mouse on top of the header.

Write the code for this problem in a new file **p3.py**.

a) Write a function called *get_csv_data*() that takes these parameters:

- *f*: an open file object in CSV format (similar to our lb-james.csv file), where the first line has only column headings, followed by lines with comma-separated data items,
- *string_pos_lst*: a list with index positions (starting from 0) for columns that have a string format. For the lb-james.csv file, this list should be [0, 2, 3, 4]. All other columns are assumed to have a *float* format. Assume that this list is not sorted in any way.
- *sep*: a string representing the column separator with default value “,”

The function *get_csv_data* returns a *nested list* we call here *data_lst,* as follows:

- it reads the first line from file *f* (with the column headings) and adds to *data_lst* a list with the column heading strings,
- after that first line, it reads file *f* line by line and parses each line based on the *string_pos_lst* parameter. All columns on a line are assumed to be *float*s if not specifically indicated in the *string_pos_lst* list. The function adds the list with parsed tokens (floats or/and strings) to the *data_lst* list. Lines with at *least one parsing error* are ignored. E.g. trying to call float(‘’) will raise ValueError; we skip this entire line and go to the next one.

Error handling: check parameters for invalid values (e.g. sep==empty string). It is acceptable for this function to throw/raise IndexError and errors related to I/O failures.

Example. Here is how this function can be used:

```
bb_file = open("lb-james.csv", "r")

bb_lst = get_csv_data(bb_file, [0, 2, 3, 4], ",")
```

(error handling is omitted here for brevity)

Element bb_lst[0] should be a list with the column headings: ["Season","Age","Tm", ……., "PTS"].

Element bb_lst[1] should be a list with column for line 1 data:

["2003-04",19.0,"CLE","NBA","SG",79.0,79.0, …….., 1.9,20.9].

Notice the types of the elements: they are strings for column indices given in list `[0, 2, 3, 4]`or floats otherwise.

b) Write in file p3.py a function *get_columns*() that returns selected columns as lists using as input argument a list returned by function *get_csv_data*().

Function *get_columns*() takes as paramers:

- *data_lst*: a list previously returned from a call to function *get_csv_data*()
- *cols_lst*: a list with the headings of columns to be returned

get_columns() returns a *nested list* having as elements a list for each column whose heading is **indicated in parameter *cols_lst***. If cols_lst==[] then the function returns []. If a column heading does not exist in *data_lst*[0], then that column is ignored.

This function is best illustrated with an example using the *lb-james.csv* file:

```
bb_file = open("lb-james.csv", "r")

james_lst = get_csv_data(bb_file, [0, 2, 3, 4])    # columns at indeces 0,2,3,4

                                                   # have a string format

selected_cols_lst = get_columns(james_lst, ["Season","Age","PTS"])
```

The returned nested list *selected_cols* then has the following data:

selected_cols_lst = [["2003-04", "2004-05", ………, "2016-17"], ← the Season column as list

[19.0, 20.0, …….., 32.0], ← the Age column as list

[20.9, 27.2, …….., 27.1]] ← the PTS column as list

(…. means we omitted many elements)

Important: in order to get credit for part b) do **NOT** use the functions from the **csv** module or other module that parses text or CSV files.

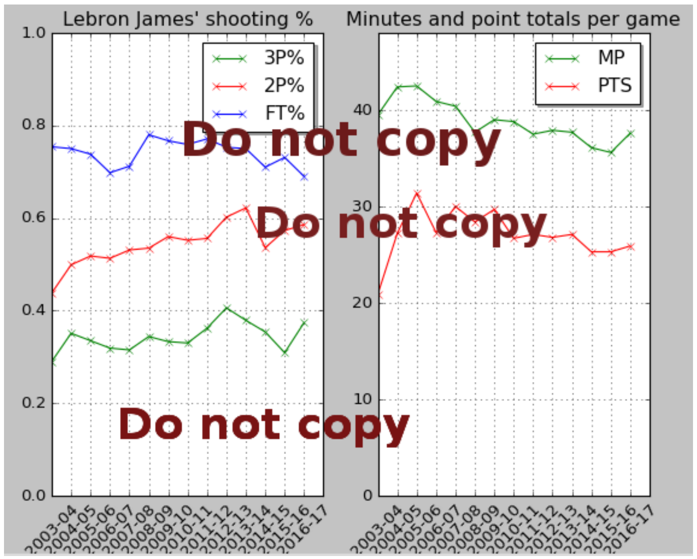
c) Now we get to LeBron. Let's use these functions to show how consistent James has been over the years.

Use these two functions and the *pylab* module to show on one pylab chart the evolution of James performance for 3-point field goal percentage, 2-point field goal percentage, and free throw percentage.

Make sure the axes and data lines are properly labeled (pylab.xlabel, ylabel, legend, title, etc.)

Write the necessary code to create the chart in file p3.py.

Here is a screenshot with the chart you should obtain. Pick the charts' layout to your choice.



Submission Instructions

- Write a heading for Problem 3 in file h3.doc.
- Insert the p3.py file in h3.doc.
- Take a **screenshot** of the required chart and paste it in the h3.doc file.

Submission Instructions

Convert the **h3.doc** file to PDF format (file **h3.pdf**) and upload the following files on Canvas by clicking on the Homework 3 link:

1. h3.pdf
2. p1.py
3. p2.py
4. p3.py

Grading:

Problem 1: 35% + 4 extra credit points

- algorithm and code correctness: 20%
- programming style: 5%

Problem 2: 30%

- algorithm and code correctness: 25%
- programming style: 5%

Problem 3: 35%

- code correctness: 25%
- programming style: 5%
- screenshot: 5%

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Only submissions uploaded before the deadline will be graded.
- You have unlimited attempts to upload this assignment, but only the last one will be graded.