LABORATORY 02-04

OBJECTIVES

- Learn how to split code into several modules that communicate via function calls
- Learn basic string and compound data type handling in Python
- Learn to specify and test your code.
- Use a more complex IDE , such as Eclipse

REQUIREMENTS

- You will be given one of the problems below to solve
- Use simple feature-driven software development process.
- The program must provide a console-based user interface that accepts the commands given exactly as they are exemplified in the problem statements.
- Use Python's built-in compound types to represent data in the problem domain.
- Iterations are scheduled for three successive labs:
    - **Iteration 1 (deadline is week 3, 25% of grade):**
        - Implement features 1 & 2
        - Use procedural programming
        - Have at least 10 items in your application at startup (so testing is easy)
        - Provide documentation
    - **Iteration 2 (deadline is week 4, 25% of grade):**
        - Also implement features 3 & 4
        - Provide documentation, specification and tests (all functions except ones in UI)
    - **Iteration 3 (deadline is week 5, 50% of grade):**
        - All required features must be implemented
        - Use modular programming (at least UI and functions modules)
        - Provide documentation, specification and tests (all functions except ones in UI)
        - This iteration will be tested by your lab professor ☺
- Data validation - when the user enters invalid commands or input values, they will be notified about the mistake.
- The documentation will contain the problem statement, feature list, iteration plan, usage scenario, work items/tasks, as shown within the *Laboratory.02-04.sampleDoc.odt* file. You can reuse the same documentation file, but you must update it accordingly.

BONUS POSSIBILITY (0.1P)

- For the last iteration (due week 5), implement an additional UI module that provides a menu-based user interface (e.g. like in example *09-CalculatorProcedural.py*). When starting the program, you will ask the user which UI they wish to use – command-based or menu-based. The program will then work accordingly.
- To receive the bonus, both UI modules must use the same underlying functions.

PROBLEM STATEMENTS

1. NUMERICAL LIST

A math teacher needs a program that will help students test different properties of complex numbers**,** provided in the ***a*+*b***i form (assume ***a***, ***b*** integers for simplicity). The program manages a list of complex numbers and allows its user to repeatedly execute the following functionalities (each functionality is exemplified):

1. Add numbers to the list.
   add *<number>*
   insert *<number>* ***at*** *<position>*

   e.g.
   add 4+2i – adds 4+2i at the end of the list
   insert 1+1i at 1 – insert number 1+i at position 1 in the list; positions are numbered from 0.

2. Modify elements from the list.
   remove *<position>*
   remove *<start position>* ***to*** *<end position>*
   replace *<old number>* ***with*** *<new number>*

   e.g.
   remove 1 – removes the number at position 1.
   remove 1 to 3 – removes the numbers at positions 1,2, and 3.
   replace 1+3i with 5-3i – replaces all the occurrences of number 1+3i with the number 5-3i.

3. Write numbers having different properties.
   list
   list real *<start position>* ***to*** *<end position>*
   list modulo [*< | = | >* ***]*** *<number>*

   e.g.
   list – write the list of numbers.
   list real 1 to 5 – writes the real numbers (imaginary part =0) between positions 1 and 5 in the list.
   list modulo < 10 – writes all numbers having modulo <10 from the list.
   list modulo = 5 – writes all numbers having modulo =10 from the list.

4. Obtain different characteristics of sub-lists.
   sum *<start position>* ***to*** *<end position>*
   product *<start position>* ***to*** *<end position>*

   e.g.
   sum 1 to 5 – writes the sum of the numbers between positions 1 and 5 in the list.
   product 1 to 5 - writes the product of numbers between position 1 and 5 in the list.

5. Filter the list.

filter real
filter modulo [*<* | *=* | *>* ***]*** *<number>*

e.g.
filter real – keep only real numbers (imaginary part =0) in the list.
filter modulo < 10 – keep only those numbers having modulo <10 in the list.
filter modulo > 6 – keep only those numbers having modulo >6 in the list.

6. Undo the last operation that modified program data.

undo – the last operation that has modified program data will be reversed. The user has to be able to undo all operations performed since program start by repeatedly calling this function.

2. CONTEST

During a programming contest for students, each contestant had to solve 3 problems (named P1, P2 and P3). Afterwards, an evaluation committee graded the solution to each of the problems solved by the contestants using integers between 0 and 10. The committee needs a program that will allow managing the **list of scores** and establishing the winners. Write a program that will implement the following functionalities (each functionality is exemplified):

1. Add the result of a new participant.

add <P1 score> <P2 score> <P3 score>
insert <P1 score> <P2 score> <P3 score> ***at*** <position>

e.g.
add 3 8 10 – add a new participant with scores 3,8 and 10 (scores for P1, P2, P3 respectively).
insert 10 10 9 at 5 – insert scores 10,10 and 9 at position 5 in the list (positions are numbered from 0).

2. Modify the scores from the list.

remove <position>
remove <start position> ***to*** <end position>
replace *<old score> <P1 | P2 | P3>* ***with*** *<new score>*

e.g.
remove 1 – set the scores of the participant at position 1 to 0.
remove 1 to 3 – set the scores of participants at positions 1,2, and 3 to 0.
replace 4 P2 with 5 – replace the score obtained by participant 4 at P2 with 5.

3. Write the participants whose score has different properties.

list
list sorted
list [*<* | *=* | *>* ***]*** <score>

e.g.
list – write the list of participants and their scores for each problem.

list < 40 – write the participants having an average score < 40.
list = 67 – write the participants having an average score = 67.
list sorted – write the participants sorted in decreasing order of their average score.

4. Obtain different characteristics of participants.
avg *<start position>* ***to*** *<end position>*
min *<start position>* ***to*** *<end position>*

e.g.
avg 1 to 5 – writes the average of the average scores for participants between positions 1 and 5 in the list.
min 2 to 7 - writes the lowest average score of the participants between positions 2 and 7 in the list.

5. Establish the podium.
top *<number>*
top *<number> <P1 | P2 | P3>*
remove [*< | = | >* ***]*** *<score>*

e.g.
top 3 – write the 3 participants having the highest average score, in descending order of their average score.
top 4 P3 – write the 4 participants who obtained the highest score for problem P3, sorted descending by that score.
remove < 70 –set the scores of participants having an average score < 70 to 0.
remove > 89 –set the scores of participants having an average score > 89 to 0.

6. Undo the last operation that modified program data.
undo – the last operation that has modified program data will be reversed. The user has to be able to undo all operations performed since program start by repeatedly calling this function.

3. FAMILY EXPENSES

A family wants to manage their monthly expenses. In order to complete this task, the family needs an application to store, for a given month, all their expenses. Each expense will be stored in the application using the following elements: ***day*** (of the month in which it was made, between 1 and 30), ***amount of money*** (positive integer) and ***expense type*** (one of: housekeeping, food, transport, clothing, internet, others). The family needs an application that provides the following functionalities (each functionality is exemplified):

1. Add a new expense into the list.
add *<sum> <category>*
insert *<day> <sum> <category>*

e.g.
add 10 internet – add to the current day an expense of 10 RON for internet.
insert 25 100 food – insert to day 25 an expense of 100 RON for food.

2. Modify expenses from the list.

remove *<day>*
remove *<start day>* ***to*** *<end day>*
remove *<category>*

e.g.
remove 15 – remove all the expenses for day 15.
remove 2 to 9 – remove all the expenses between day 2 and day 9.
remove food – remove all the expenses for food from the current month.

3. Write the expenses having different properties.

list
list *<category>*
list *<category>* ***[*** *< | = | >* ***]*** *<value>*

e.g.
list – write the entire list of expenses.
list food – write all the expenses for food.
list food > 5 - writes all expenses for food with an amount of money > 5.
list internet = 44 - writes all expenses for internet with an amount of money = 44.

4. Obtain different characteristics of sublists.

sum *<category>*
max *<day>*
sort *<day>*
sort *<category>*

e.g.
sum food – write the total expense for category food.
max day – write the day with the maximum expenses.
sort day – write the total daily expenses in ascending order by amount of money spent.
sort food – write the daily expenses for category food in ascending order by amount of money spent.

5. Filter the list of expenses.

filter *<category>*
filter *<category>* ***[*** *< | = | >* ***]*** *<value>*

e.g.
filter food – keep only expenses in category food.
filter books < 100 – keep only expenses in category books with amount of money < 100 RON
filter clothing = 59 – keep only expenses for clothing with amount of money = 59 RON

6. Undo the last operation that modified program data.

undo – the last operation that has modified program data will be reversed. The user has to be able to undo all operations performed since program start by repeatedly calling this function.

4. BANK ACCOUNT

John wants to manage his bank account. In order to complete this task, John needs an application to store, for a given month, all the bank transactions performed on his account. Each transaction is stored in the application using the following elements: ***day*** (of the month in which the transaction was made, between 0 and 30), ***amount of money*** (that was transferred, positive integer), ***type*** (in - into the account, out – from the account), and ***description***. Help John by creating an application that provides the following functionalities (each functionality is exemplified):

1. Add new transactions to the list.

add *<value> <type> <description>*
insert *<day> <value> <type> <description>*

e.g.
add 100 out pizza – adds to the current day an out transaction of 100 RON with the "pizza" description.
insert 25 100 in salary – insert to day 25 an in transaction of 100 RON with the "salary" description.

2. Modify transactions from the list.

remove *<day>*
remove *<start day>* ***to*** *<end day>*
remove *<type>*
replace *<day> <type> <description>* ***with*** *<value>*

e.g.
remove 15 – remove all transactions from day 15.
remove 5 to 10 – removes all transactions between day 5 and day 10.
remove in – remove all the in transactions from the current month
replace 12 in salary with 2000 – replace the amount for the in transaction having the "salary" description from day 12 with 2000 RON

3. Write the transactions having different properties.

list
list *<type>*
list [*< | = | >]* *<value>*
list balance *<day>*

e.g.
list – write the entire list of transactions.
list in – write all the in transactions.
list > 100 - writes all transactions having an amount of money > 100.
list = 67 - write all transactions having an amount of money = 67
list balance 10 – computes the account's balance on day 10. This is the sum of all in transactions, from which we subtract out transactions occurring before or on day 10.

4. Obtain different characteristics of the transactions.

sum *<type>*
max *<type> <day>*

e.g.
sum in– write the total amount from in transactions
max out 15 – write the maximum out transaction on day 15.

5. Filter.

filter *<type>*
filter *<type> <value>*

e.g.
filter in – keep only in transactions.
filter in 100 – keep only in transactions having an amount of money smaller than 100 RON.

6. Undo the last operation that modified program data.

undo – the last operation that has modified program data will be reversed. The user has to be able to undo all operations performed since program start by repeatedly calling this function.

5. APARTMENT BUILDING ADMINISTRATOR

Jane is the administrator of an apartment building and she wants to manage the monthly expenses for each apartment. Jane needs an application to store, for a given month, the expenses for each apartment. Each expense is stored in the application using the following elements: ***apartment*** (number of apartment, positive integer), ***amount*** (positive integer), ***type*** (from one of the predefined categories: water, heating, electricity, gas, other). Help Jane by creating an application that provides the following functionalities (each functionality is exemplified):

1. Add a new transaction to the list.

add *<apartment> <type> <amount>*

e.g.
add 25 gas 100 – add to apartment 25 an expense for gas in amount of 100 RON.

2. Modify expenses from the list.

remove *<apartment>*
remove *<start apartment>* ***to*** *<end apartment>*
remove *<type>*
replace *<apartment> <type>* ***with*** *<amount>*

e.g.
remove 15 – remove all the expenses of apartment 15.
remove 5 to 10 – remove all the expenses from apartments between 5 and 10.
remove gas – remove all the expenses for gas from all apartments.
replace 12 gas with 200 – replace the amount of the expense with type gas for apartment 12 with 200 RON.

3. Write the expenses having different properties.

list
list *<apartment>*
list [*< | = | >* ***]*** *<amount>*

e.g.
list – write the entire list of expenses.
list 15 – write all expenses for apartment 15.
list > 100 - write all the apartments having total expenses > 100 RON.
list = 17 - write all the apartments having total expenses = 17 RON.

4. Obtain different characteristics of the expenses.

sum *<type>*
max *<apartment>*
sort apartment
sort type

e.g.
sum gas – write the total amount for the expenses having type "gas".
max 25 – write the maximum amount per each expense type for apartment 25.
sort apartment – write the list of apartments sorted ascending by total amount of expenses.
sort type – write the total amount of expenses for each type, sorted ascending by amount of money.

5. Filter.

filter *<type>*
filter *<value>*

e.g.
filter gas – keep only expenses for "gas".
filter 300 – keep only expenses having an amount of money smaller than 300 RON.

6. Undo the last operation that modified program data.

undo – the last operation that has modified program data will be reversed. The user has to be able to undo all operations performed since program start by repeatedly calling this function.