**General Instructions**:

1. **Submission date: 25.8.2016**.

You are advised to at least start with the exercise before the midterm to gain some practice.

1. You should read through all the instructions before starting to code.
2. The solution should include two files: contact.py and phonebook.py.
3. Write your name and ID in a comment at the top of the files.
4. Make sure your code runs smoothly with Python 3.4.x or higher.
5. You can submit in singles, pairs, or triplets.

If you submit in pairs or triplets, since we use the Moodle system, then each student should submit the files by himself, and all students submitting together should include all their names and IDs in the comment at the top of the files.

1. You should test all the requested functionalities.

A sample run output is provided at the end of this document.

1. **Write readable code**: use meaningful names, don't overcomplicate things.

Working code that is overcomplicated will lose (some) points.

1. **Use object oriented principles**: Reuse code on inheritance, base classes should not be aware of their sons, same goes with sibling classes.

**Exercise Details**:

In this exercise you will write a program simulating a contacts app, similar to the one in your smartphone.

There are 4 types of contacts;

Each should be implemented in its own class (all in contact.py), with the following included fields:

**Contact – consists of name and cellphone.**

**FriendContact – extends Contact by adding homePhone and personalEmail.**

**ProfessionalContact – extends Contact by adding workPhone and workEmail.**

**ProfessionalFriendContact – combines FriendContact and ProfessionalContact.**

**In order to save space, all fields but the name field are optional, so all the above mentioned data attributes will be added to an instance of its class only when needed**.

The phone book itself will be implemented by a class called *PhoneBook* (on phonebook.py) which will have a list of contacts.

It will also have a method called *Start()* that will run an infinite loop that prints the following menu:

**What would you like to do?**

**1 - Add a new contact**

**2 - Show all contacts**

**3 - Edit a contact**

**4 - Find a contact**

**5 - Delete a contact**

**6 - Exit**

**-->**

You should read the user's command from the console and check whether it is legal (1-6).

If it is not, you should print a mistake message and read the command again.

**Command 1** displays the following question:

**Should this contact be Simple (S), Friend (F), Professional (P) or Both (B)?**

You should check that the answer is legal (S, F, P or B), and read again if necessary.

According to the answer, you should create the appropriate contact,

read its fields one by one from the user, and add it to the contacts list.

You should allow empty inputs to any of the fields but the *name* field.

For non-empty fields, phone numbers should consist of digits only and emails should include '@' and a dot ('.'). You can use the string methods *isdigit*() and \_\_*contains*\_\_() to check for validity.

**Command 2** prints all the contacts, along with their index on the list.

The first index should be 1 and not 0 so it will be more readable to the user.

You should print the contacts by calling *print*() on each member of the list, so you need to implement \_\_*str*\_\_().

**Print only the existing fields**, along with the field name (e.g. Name: Avi).

The list should always be **sorted** according to the contact's name.

To guarantee a fast user response you should not sort the list just before printing it;

at the time of printing, it should have already been sorted, i.e. it should be kept sorted at all times.

**Command 3** prints the following line:

**Enter a valid number of the contact you wish to edit:**

The user should insert the contact's number according to the printing result of command 2 or command 4.

You should check that it's a valid contact number, and print a mistake and read again when necessary.

Then, you should print the same line from command 1:

**Should this contact be Simple (S), Friend (F), Professional (P) or Both (B)?**

After reading the user's choice, print the following lines:

**For the following fields**

**click enter if there's no change,**

**a new value if you want to replace the field,**

**or x if you want to delete the field (the name field cannot be deleted).**

As in command 1, you should create the appropriate contact, read its fields one by one from the user, and place it in the contacts list, replacing the existing contact.

Unlike command 1, you should also print the existing values in parentheses when you prompt the user for input, and allow him to insert a blank input that will **keep the current value**.

For example, the following keeps the existing name untouched and replaces the cellphone from 123 to 456:

Name (Avi Cohen):

Cell Phone (123): 456

The user can also type x to **completely** delete the field.

If x is given on a non-existing field, then nothing should happen regarding that field

(you should not store x as the value).

Notice that the user can decide on a different contact type and you should prompt only the relevant fields.

For example, if the contact was a *ProfessionalContact* and is replaced to a *FriendContact*, then only the *name* and *cellphone* fields are prompted (with their existing values), *workPhone* and *workEmail* will not be prompted (they will be lost anyway), and the not-yet-existing fields *homePhone* and *personalEmail* should be prompted for input.

**Command 4** prints

**Type contact details (name, phone, email):**

Then it will look for the input from the user at all the fields of all the contacts **in the list(**in phonebook, where each ‘Contact’ has the ‘Match’ method**)**, and will print the ones that contain the string in any of their fields.

As is command 2, the print will include the contact number in the list, and not in the sub lists of matching contacts, i.e. if the string matches contacts in indices #4 and #7 it will print 5 and 8 respectively.

This way, if the user chooses to edit one of those contacts, he'd use the printed contact number.

**Command 5** prints

**Enter a valid number of the contact you wish to delete:**

Then it checks whether the contact number is valid, and deletes the contact.

**Command 6** exits the infinite loop (which actually terminates the program).

**Contact Classes Instructions**:

All the contact classes should have the following methods:

\_\_*init*\_\_(),

\_\_*lt*\_\_(),

\_\_*str*\_\_(),

*ReadValues*()

*Match*()

\_\_*lt*\_\_() is responsible for contact comparison so the contacts' list could be sorted.

\_\_*str*\_\_() is responsible for printing and allows us to call print(<instance of contact>).

*Match*() takes a string and returns **true** if the string appears as a substring (or fully) in any of the contact's fields.

\_\_*init*\_\_() should be signed like**: def** \_\_init\_\_(self, olderContact = **None**):

This means that it *can* get an optional argument *olderContact*

(If we create an instance without passing this argument, then its default value is None).

If we do pass an *olderContact* argument, it means that the contact is being created in order to replace an older contact, and we should read and save the appropriate fields from that older contact.

Note that the older contact can be of any contact type.

On each derived class of *Contact*, you should check whether *olderContact* is of the current type in order to read the fields.

Even if it is of the current type, it is not guaranteed that all (optional) fields do exist, and you should also check whether they exist or not.

You can use *dir*() and *count*() , or *hasattr*() to do so.

In order to save place you should not create the optional fields unless they appear in *olderContact*.

*ReadValues*() prompts the user and reads the appropriate values for the fields.

If a field already exists, the current value should be prompted.

For an empty input, if the field exists then the value remains untouched, and if it does not exist, don't create it.

You should delete an existing field (except for *name*) if x was provided.

**Important Notes**:

1. **There should be a clear separation of responsibilities between different objects:**
   1. ***PhoneBook* should not read the different contacts' fields.**

**It is responsible for creating the contacts and "telling" them to read their values.**

* 1. **Each type of contact is responsible for reading /printing its own fields, and only its own fields.**
  2. **Avoid code duplications; wherever possible, use the parent class' implementation and extend it.**

1. **Base classes should not be aware of any derived classes or their fields.**
2. **Do not change the signature of *\_\_init\_\_()* and *ReadValues()*, so \_\_*init*\_\_ should only get the *olderContact* argument and *ReadValues* should not get any argument.**

**Multiple Inheritance Instructions**:

Obviously you would want to reuse the code of base classes in the derived classes.

On the *ProfessionalFriendContact* class, if you initialize the class using both its parents' constructors, which both also initialize the basic fields of *Contact* *(name* and *cellphone)*, then those fields will be initialized twice.

This is somewhat redundant, but not a big deal.

Same goes with finding a string in *Match*(), the string will be searched in *name* and *cellphone* twice.

On the other hand, calling *ReadValues*() or \_\_*str*\_\_() in both the parents will cause undesired behavior: In the first, the user will be prompted for the *name* and *cellphone* fields twice, and in the latter those fields will be printed twice.

As we learned in class, there are two options to avoid this:

The first is the "non-Pythonic" way of splitting methods in *ProfessionalContact* and *FriendContact* to two different methods and on *ProfessionalFriendContact* only call the necessary methods of its parents.

The second is the "Pyhtonic" way of using *super*().

In this exercise, implement *ReadValues*() in the Pythonic way and \_\_*str*\_\_() in the non-Pythonic way.