Contacts App Project

Purposes of this assignment

- Practice with lists and tuples
- Practice with logic, functions, and general problem solving
- Practice pair programming

General Idea

In this assignment you will create a text-based "Contacts App", very much like what is in your cell phone. The project itself consists of two versions. **`contacts-one.py`** is the simpler version, since you'll only be working with contact phone numbers. **`contacts-two.py`** introduces 2D data structures (i.e., a collection of collections), which allow you to store additional data about contacts, such as their names.

I suggest you complete each function in the order they appear here. The very last line of each of your files should be a single call to the **`run()`** function.

---

contacts-one.py

At the top of this file, you should type all of the following:

```
# Phone Contacts App v1.0 (or whatever you want to call this)
# By: Your name and your partner(s) names
# SMASH CS2 (year)

from random import randint

contacts = []
```

And now for the rest.

check_number(number)

This is a *helper function*, or a function whose only purpose is to be called by other functions in order to eliminate redundancy. It's generally a good strategy to create these kinds of functions when you find yourself repeating the same lines of code over and over across your program.

This function returns **`True`** if the input number is valid, and **`False`** otherwise.

The number should be expressed as a positive 10-digit integer (i.e., area code + the rest). Assume no country codes, parentheses, decimals, or other special characters will be passed along with input.

However, the function should be sure to consider negative numbers as invalid. Also, leading zeros are considered "blanks", meaning **0234** is actually 3 digits.

Hint: example inputs
valid: 1234567890
invalid: 0000000001
invalid: 1234
invalid: -1234567890
invalid: 99999999999999999

Hint: determining the number of digits in a number
You can divide a number by 10 using the integer division operator "//", which truncates (cuts off) the last digit of a number. How many times you need to truncate the last digit of a number until you reach 0 tells you how many digits are in that number.

Example:
How many digits are in the number **1234**?

```
1234 // 10 -> 123
123 // 10 -> 12
12 // 10 -> 1
1 // 10 -> 0
```

Total operations: **4** (So 1234 is a 4-digit number)

add_contact(number)

This function adds a contact's phone number to **contacts** while preserving that list's numeric order (least to greatest). Note that because **contacts** exists outside of all functions (you declared and initialized it at the top of the file), its scope is considered global within this file, meaning when I say **contacts** I am talking about the same list, no matter where I am in the file.

First, this function should check if the number entered is valid by simply calling the `check_number()` function you wrote above. If the number is not valid, this function should print **"Invalid number"** and do nothing else.

Otherwise, this function should then check if the length of your contacts is zero. If so, the number is simply appended (or "added to the end") of the list.
Finally, assuming the list is not empty, the function should insert the number where it belongs in **contacts**. That is, you will need to loop through **contacts** to find the appropriate position where the number belongs, and **insert** it there. No, you may not simply append the number and then use sort(). That makes it too easy. ^_^

In either case, if the number was successfully added to contacts, you should print "Contact added". If the number already exists, you should print "Contact already exists".

`remove_contact(number)`

This function removes a contact's phone number from **`contacts`** while preserving that list's numeric order (least to greatest).

Much like the above, you should first check if the number entered is valid. Second, if the contacts list is empty, you should print **`"You have no contacts"`** and do nothing else.

Otherwise, you should loop through the **`contacts`** until you either find the number requested or you reach the end of the list. If you find the number, you should pop() that number, which removes it from the list, and print **`"Contact removed"`**. If you reach the end of the list, which means you did not find the number, you should print **`"Contact not found"`** and do nothing. And no, you may not simply use "remove(number)" since, again, that makes it too easy. ^_^

`print_contacts()`

This function simply prints all of the numbers that are currently in **`contacts`**, one by one. If **`contacts`** is empty, it prints **`"You have no contacts"`** and does nothing else.

`test_add_contact(num_tests)`

This function tests your add_contacts() method with random valid numbers, depending on how many tests you specify as input (i.e., you'll need a loop). Once all the numbers are added, it calls print_contacts(). The only purpose of this function is for the programmer to test her own code. You'll find yourself doing lots of these as you learn more programming.

To generate random numbers, you'll need to call the function **`randint(a, b)`**, which we imported from the **`random`** library at the top of the file. According to the API, **`randint(a, b)`** "returns a random integer N such that a <= N <= b".

A brief word about "random" numbers and computers

These numbers are not "truly" random but pseudo-(i.e, fake) random. This is because the nature of what we call "randomness" cannot be computed by a "deterministic Turing machine", which is exactly what a computer is. Therefore, we generate numbers that *behave* as if they were random, based on something called a "seed", or some unique number used to help calculate a pseudo-random number. The most common seed is your computer's system time. Just think, today's date is a truly unique value that will likely never occur again (e.g., May 17, 2017 at 10:47pm will never, ever happen again).

run()

This runs your program by executing a loop that prompts the user for input within the console.

When the program starts, the following is printed:

```
Phone Contacts (v1.0)
Please choose a number.
1: Add contact
2: Remove contact
3: View all contacts
4: Quit
>
```

The user enters a number representing what they wish to do. If they enter an invalid choice, you should print "Invalid choice" and return to the selection menu above.

Otherwise, you simply call the functions you wrote above, corresponding to the choice the user has made. and return to the selection menu.

The program ends when the user enters 4, at which you should print a goodbye message.

Hint: getting keyboard input
In order to get and store keyboard input, you should use this:

```
choice = input()
```

Note that if you place anything inside **input()** yourself, that text becomes part of the prompt, in which case anything the user types will appear after it on the same line. So you if you wish to be fancy you can give the user a prompt like this:

```
choice = input("> ")
```

The variable **choice** now contains whatever the user entered. Note that this data is a **string** type, which means checking a phone number will require casting that string to an integer, otherwise your code will break. Here is an example of casting:

x = int("1234")

In the above example, the string **"1234"** is cast as an **int** and stored in **x**. So, **x** contains **1234** (no string quotes) and can be treated like a normal number.

---

contacts-two.py

Time for some improvements! Before you attempt this part, be sure you have had some time to practice working with tuples with your teacher and classmates.

First, go ahead and copy all the code you wrote in the first part and paste it here. The top of the file should look the same, only be sure to change the title to something like "Phone Contacts App v2.0".

Complete the following functions in order. Some functions will need to be updated and are marked as such. Likewise, new functions will be marked accordingly.

Update: add_contact(firstname, lastname, number)

This function adds to **contacts** a tuple that better represents the way we think of a "contact" in our phones. You know, that person's NAME! ^_^ In other words, the tuple is the "contact", while each value within is a piece of info about the contact.

Both names must contain only letters (no spaces, no special characters, no numbers, etc.) and may not be empty. Numbers must still be valid, using the same check_number() method as before.

Mostly, this function should operate just like before, except a few differences:

- Now you are adding a tuple to **contacts** and not just the number.
- In terms checking if a contact already exists, you are now checking for the first and last names, rather than the number (imagine people that might share the same phone number, like with an office phone). For the sake of this project, no two contacts may share the same first and last names, though they can share the same first or last names.
- Your function should now preserve the alphabetical order of the **contacts** list by each contact's first name. Just like in real life!

Finally, do not worry about hyphenated last names. You may assume that anyone with a hyphenated last name will simply have their name "camel cased" before passing it into the function (e.g., "Barba-Hernandez" would be passed as "BarbaHernandez").

Hint: checking if a string contains only letters
To check if a string contains only letters, you can use the isalpha() function, which is part of the string library.

```
x = "abc"
y = "123"
z = "  a"
x.isalpha() -> True
y.isalpha() -> False
```

z.isalpha() -> False

Hint: comparing identical strings with different cases
Just as before, you don't want to add a contact to your list if it already exists. But what if the name "Osciel" is already in your list, and someone tries to add the name "osciel"? We should tell them the contact already exists. However, doing a simple string comparison wouldn't work.

"Osciel" == "osciel" -> False

To make this work, we can try this cool trick. The string library also has functions called lower() and upper(). These work in the following ways:

"Osciel".lower() == "osciel" -> True
"YabbaDabbaDoo".upper() == "YABBADABBADOO" -> True

I recommend using one of these functions to check for existing contacts.

Hint: alphabetical order of strings
To check if one string is alphabetically before or after another, you can simply compare them, just as you would with numbers. Just be sure to also use lower() or upper() from above on each string before comparing.

"bAby".lower() < "Beluga".lower() -> true
"Arthur".lower() > "aArOn".lower() -> true

New: remove_contact_name(firstname, lastname)

Just as it says, this removes a contact based on that contact's first and last names.

Otherwise, this function operates just as **remove_contact()**, with all the necessary print statements to communicate to the user what is going on.

Obsolete: remove_contact(number)

We no longer need this, since we have a better function (see above). Normally we would simply comment it out just in case we would like to use it later. But we already have an earlier version of this project saved, so we don't have to worry.

Update: print_contacts()

This function should be updated to print each contact's first name, last name, and phone number, in that order, line-by-line. Otherwise nothing else changes.

Update: run()

This function should be updated to reflect the changes we made above. Namely:

- Adding a contact is now a series of three prompts, each asking for its own input (i.e,. first name, last name, and phone number). Once you have all three you can pass them to **`add_contact()`** for processing.

- Removing a contact should now require a first and last name, not a number.

Your selection menu should display the 2.0 status of this app. ^_^ Be proud! You did all this from scratch… bugs and all! @_@

And that's it! Play with your completed app and see what additional improvements you might make.

One thing you'll notice is when a person enters multiple invalid values while trying to add a contact, the program will only tell them about the top-most invalid value. For example, entering "1234, 482, 3" as input will only tell you that 1234 is invalid. This is exactly the way we programmed it, so it isn't anyone's fault but ours. How might you fix this so that the user knows about the other two inputs as well?

Hmm… how about adding other details to the tuple, such as email address? How would you check if an email address is valid or not? Maybe too difficult to do right now, but it's at least worth a shot. You never know what you can do until you try! And don't forget to use those APIs!

Hey, that rhymed, didn't it? Bars! ^_^