CSCI 141 Computational Problem Solving Project 2

Due Friday October 14, 2022 at 2200

Random digit dialing is a process commonly used in surveys to try and avoid bias in selecting participants (and it wouldn't be surprising if robo-calls used it too). A lot of random digit dialing is not completely random. For example, if we consider phone numbers in the United States, a subset of area codes would be selected in advance to represent a sampling area, and then random phone numbers would be generated within that area code. In this project, we are going to explore how to generate random phone numbers that are valid residential or cellular numbers, i.e., you should reach someone when you call!

Phone Numbers in the United States

Phone numbers in the United States have 10 digits, for example: 757-221-4000. The first three digits are the **area code** (757 in this example) which indicates a specific geographic area. At the time this project was written, there are 338 unique area codes used in the US for residential and cellular numbers (when we consider only states). The fourth through sixth digits are known as the **prefix** (221 in this example). Prefixes for residential and cellular numbers can be anything except 555, 958, 959 and x11, e.g. 211, 511, 811, 911 (anything with 11 as the last two digits). There are no constraints on the last 4 digits of the phone number (4000 in this example). The last 4 digits are called the **suffix**.

Assignment

Your task is to create 4 functions and write a short main program that uses these functions. We have provided two skeleton files: **Project_2.py** for your function code, and **Project_2_Main.py** for your main program. The **Project_2.py** file contains an import line for the random module and a list called AREA_CODES which contains all of the valid area codes in the United States. It also contains two lines that must be the start of your program.

You are required to follow the specifications below to complete the project. Read them carefully. If you choose to complete this project in some other way, you may receive no credit for your work.

Function One: make_prefix()

Write a function called **make_prefix** that takes no arguments and **returns** a string of 3 randomly selected digits that is a valid prefix. according to the rules explained in the section above. The prefix is the part of the phone number that lies after the area code. For example, in the number, 757-683-5616, the prefix is 683. **The rules for what is valid are given in the Phone Numbers in the United States section**

above (the prefix is not 555, 958, 959, or x11 (last two digits 11, e.g. 911)). **make_prefix** always **returns** a 3 digit string, for example: `'834'`

You have been given the first two lines of code for this task. These lines select the first digit for the prefix and the second digit of the prefix respectively.

If you choose not to use these variables, you will receive no credit for this part of the assignment.

The variables **first** and **second** will be strings of length 1. They could be for example: **first** `'7'` **second** `'3'`

To complete the function, you need to select the third digit of the prefix in such a way that the prefix will be **valid**. This depends on what the first two digits are. The best way to complete this is to write out the logic on paper. Here's something to get you started:

If **first** is `'5'` and **second** is '5', what are the possible choices for the third digit of the prefix? What digits should be excluded when choosing the third digit?

Once you've worked out the logic for selecting the third digit, the last task is to put the digits together to make the prefix. The prefix should be stored in a variable. In your final product, it should not be printed, but you may want to print it for debugging purposes.

Use good programming style. What do we mean? Don't have extraneous conditions. For example, you shouldn't need a separate condition for the case of 958 and 959, those can be handled in the same place.

Function Two: make_suffix()

The function **make_suffix** takes no arguments and **returns** a 4 digit string by selecting digits randomly, for example: `'9824'`

As explained above, the suffix is the part of the phone number that lies after the prefix. For example, in the number, 757-683-5616, the suffix is 5616.

The second function will generate a random suffix. There are no rules for suffixes, so you just need a string of 4 random digits.

There are several ways to do this, but we will use a **for** loop. You have been given code for function 2 that does not work correctly. Debug this code.

For debugging, you may not add, delete, or move lines. The structure should stay the same. The changes for the most part are minimal in terms of what you type, but large conceptually.

Write out the logic on paper first. Here's a few things to think about: What is each variable supposed to represent (try printing each variable if you have no idea)? What type is each variable? Are the right variables being used for each operation? What is each line of the code you were given trying to do?

To complete function 2, edit the code given to you to create a 4 character string that is a suffix.

Function Three: make_phone_number(area_codes = AREA_CODES, sep = '-')

The function **make_phone_number** creates a randomly generated valid 10 digit phone number. The optional argument **area_codes** is a list of area codes (as strings) for which the user wants to generate a phone number. Notice that the default value is the list AREA_CODES. The optional argument **sep** provides a character used to separate the area code, prefix, and suffix.

make_phone_number uses the **make_prefix()** and **make_suffix()** functions to generate the prefix and suffix respectively. It should combine these with an area code chosen from **area_codes** and return the phone number as a 12 character string in the format: three digits, separator, three digits, separator, four digits. The optional argument **sep** controls what is used to separate the digits. The default value of sep is a dash: -. For example, the return value for **make_phone_number** with the default value of sep could be `'757-221-3455'`. The user could pass in a different character for **sep**, for example, `'*'`. The return value could be, for example, `'789*555*3455'`. Remember that the actual digits will be randomly selected, so you can expect to get a different phone number each time you run the function.

The point of having the **area_codes** argument is to allow the user to limit phone numbers to a specific set of or perhaps even one specific area code. For example, they could pass in `['757', '804']`, which means that the phone number generated will either start with 757 or 804 (it only chooses an area code from that list).

Key points:

- This function MUST use your **make_prefix** and **make_suffix** functions in order to receive full credit. It should not repeat the code from those functions.
- Write down the steps you need on paper.
- You can assume that if a value is provided for **area_codes**, it will be a list containing at least one string. You can assume that the input **sep** if provided will always be a string that is a single character. Remember that this function should return a 12 character string.

Function Four: hampton_roads_number(tester, sep = '-')

Beginning May 9, 2022, new telephone lines or services in Hampton Roads may be assigned numbers with the new 948 area code in addition to 757. You will write a function called hampton_roads_number. This required parameter **tester** is a string. The optional parameter **sep** is also a separator that separates the area code, prefix, and suffix; its default value is a dash: -. This function checks if the tester is a valid phone number in Hampton Roads District.

This function returns a **Boolean**: True if the tester is a valid phone number in Hampton Roads District with the separator given by the arguments, and False if it does not. This function should print nothing. The user will not pass in arguments of an incorrect type.

Key points:

- A valid phone number in Hampton Roads District must: (1) have 10 digits; (2) either start with 757 or 948; (3) be in the format: three digits, separator, three digits, separator, four digits.
- You can assume that the input **sep** if provided will always be a string that is a single character.
- You can assume that the prefix of the tester will not be 555, 958, 959, or x11.

Here are some examples:

hampton_roads_number('757*819*1111', '*')
would return: True

hampton_roads_number('767-819-1i11', '*')
would return: False

Main Program: Generate Phone Numbers

You will write a brief main program that makes use of your functions. This must be saved and submitted in **Project_2_Main.py**.

The main program will make use of your **make_phone_number** function. You will be graded specifically on coding style for this part of the project – that means writing efficient code that is not repetitive, and does not specify default arguments whenever possible. The main program should do the following:

- Generate 5 random phone numbers with a while loop, using the full list of area codes (AREA_CODES). Use the default separator. These numbers should be stored in a list. After the full list is generated, print it.
- Generate 5 random phone numbers with a for loop, using the full list of area codes (AREA_CODES). Use an asterisk (*) for the separator. These numbers should be stored in a list. After the full list is generated, print it.
- Generate 10 random phone numbers with a while loop, for the area code 937 only. Use comma (,) as the separator. These should be stored in a list. After the full list is generated, print it.
- Generate 10 random phone numbers with a for loop, for the area code 503 only. Use the default separator. These should be stored in a list. After the full list is generated, print it.

There should be no def lines in your main program file.

SUBMISSION EXPECTATIONS

Project_2.py Your implementations of **make_prefix, make_suffix, make_phone_number,** and **hampton_roads_number**. Do not change the names of parameters in the function def lines or the names of the functions. **No additional code including input lines, print lines, and function calls should be submitted. Make sure that you have the correct import lines.**

Project_2_Main.py Your implementation of the main program as specified. Your functions must not be repeated in this file. They must be imported using the correct import line. **There should be no input lines. Make sure that you have the correct import lines.**

Project_2.pdf A PDF document containing your reflections on the project. **You must also cite any sources you use. Please be aware that you can consult sources, but all code written must be your own.**

POINT VALUES AND GRADING RUBRIC

make_prefix: 19

make_suffix: 18

make_phone_number: 18

hampton_roads_number: 18

Main Program: 20

Write-up: 1.5 points

Autograder: 5.5 points

N.B.: Your code should be concise and efficient. For example, you should not write extraneous structures, should not specify default arguments, should get rid of unnecessary extra variables, and should comment out debugging lines.

You must use the structures we have learned in this class to complete this assignment.

Implementations using structures we have not covered may receive no credit. This means NO LIST AND STRING METHODS.

You should not be using list comprehension either – this is a common solution you will find on the internet. Use what we have covered in class for Module 2.

References:

1. https://www.13newsnow.com/article/tech/757-hampton-roads-virginia-new-area-code-948.

RUBRIC FOR MANUAL GRADING

Grade Level	**Appropriateness**	**Style**	**Code Efficiency**
A	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; any references used cited in write-up	Follows Python style guidelines we have discussed to include appropriate naming of variables and spacing; Proper use and naming of constants	Only contains structures necessary to accomplish the task at hand; Uses the most concise implementation possible
B	Code uses structures and modules discussed in class; variable names meaningful; references cited in write-up	Largely follows Python style guidelines but may be a few minor deviations	Only contains structures necessary to accomplish the task at hand; Implementation is generally concise
C	Code uses structures and modules discussed in class with minor additions from outside sources (e.g. string formatting) appropriately cited; most variable names meaningful	Several deviations from Python style guidelines; constants not used properly	Contains a small number of unnecessary structures such as casts that are not needed; Implementation somewhat concise
D	Code uses some structures and modules taken from outside sources with citations; variable names and code structure hard to decipher	Only loosely follows Python style guidelines	Contains a large number of unnecessary structures such as unused variables and casts; Implementation somewhat concise but could be improved, code may be hard to follow
F	Code uses structures and modules largely from outside sources only; variable names hard to decipher; code written in other programming languages or Python 2.7	Code appears to have been written with no consideration for style	Contains many unnecessary structures; Code is difficult to follow and overly verbose