Lab Assignment 2: Steganography, Detecting and Decoding a Secret Message

The Five Eyes alliance links Canada's intelligence services to the United States' National Security Agency (NSA). In an NSA museum lecture, a Director of Education reveals how *steganography* has been used over millennia to hide information in plain sight. This lab assignment concerns the determination of a secret message, about an international rescue, from a list of university phone numbers. Each student writes a script to detect if a number likely represents a message and, if so, to decode the secret.

Version 0: Get Started

Unzip `V0GetStarted.zip` into your Working Directory. In the Files pane, look for `decode.py`, a text file of Python code, and `decodeTestAB_v0.txt`, a text file of iPython Console side effects. Open `decodeTestAB_v0` in the Editor. This file is a *storyboard*, a plan of end user interactions, for two test cases when the script, `decode`, runs. It shows prompts, sample inputs, and desired outputs.

Open `decode` in the Editor. Review the script section called `PARSE INPUT`. Without the `dtype=int` argument, the `np.array` expression would return a NumPy array of individual symbols, or characters, from a number entered by the end user. With the argument, it returns an array of integer digits.

Select `Run >> Run` from the menu to run, or execute, `decode`. Enter the input and check the output, in the Console, following the storyboard, `decodeTestAB_v0`, for `TEST CASE A`. In the Variable Explorer, click the trashcan icon to "Remove all variables"; or, in the Console, enter `%reset -f`. Run the script a second time. Enter the input and check the output following the storyboard for `TEST CASE B`.

As you will edit `decode`, save a copy as `decode_v0.py` (use `File >> Save copy as...`) to have an earlier version with no syntax or runtime errors on given test cases. Before you submit Version 1 or 2 of your program, rename the file as the lab instructor requires, e.g., add your CCID to the filename.

Version 1: Partial Decode

Unzip the files, `decodeTestAB_v1.txt` and `decodeTestCD_v1.txt`, in the `V1PartialDecode.zip` file into your Working Directory. When you complete Version 1 sufficiently and try each given test case, side effects of your `decode` program should match `decodeTestAB_v1` and `decodeTestCD_v1`.

Create a new script section, `RULES 1 TO 4`, below the last one, `PARSE INPUT`. Modify the program to implement two rules, Rules 1 and 2, to detect if an entered number *may* represent a valid message and two rules, Rules 3 and 4, to partially decode the message. Put your code under `RULES 1 TO 4`.

Rule 1: A valid number must have exactly eleven digits. If the number entered is invalid, print the message "`Decoy number: Not eleven digits`" to the Console, as shown in a test case.

Rule 2: A valid number must pass two digit-sum tests. If the sum of the leftmost five digits is even and the sum of the rightmost five digits is even, the number *may* be valid. Otherwise, the number is invalid. If the number is invalid this way, print the message "`Decoy number: A digit sum is odd`".

Rule 3: Calling the leftmost digit the 1st digit, multiply the 9th digit by the 8th digit and then subtract the 7th digit to get the rescue day number. Print the day number, in the format shown by a storyboard.

Rule 4: If the 6th digit is a multiple of 3, the rescue place number is the 11th digit minus the 10th digit. Otherwise, the place number is the 10th digit minus the 11th digit. Print the number, as shown.

Rules 1 and 2, when broken, result in a "`Decoy number`" message. Otherwise, as shown, output all digits before day and place numbers. When a Rule is broken, there must be no output from subsequent Rules. Functionally, what variables exist, and their values, do not matter when a correct program ends. Non-functionally, avoid computing Rules where possible while also considering readability.

Do not use repetition (`for`/`while` loops) and/or invocation (user-defined function) program flows. Also, do not use `exit` statements of any kind. Execution begins, syntax errors aside, at the first non-blank non-comment line and must end, runtime errors aside, at the last non-blank non-comment line. Given test cases are a small subset of possible test cases. Compose and test additional cases yourself.

Under `PARSE INPUT` and `RULES 1 TO 4`, write comments to summarize what the section does, or is supposed to do, in your own words. Complete the program's initial header, attending to names of and percentages from others. Submit your Version 1 solution, `decode` only, by the Version 1 deadline.

Version 2: Full Decode

Unzip the files, `decodeTestAB_v2.txt` and `decodeTestCD_v2.txt`, in the `V2FullDecode.zip` file into your Working Directory. When you complete Version 2 sufficiently and try each given test case, side effects of your `decode` program should match `decodeTestAB_v2` and `decodeTestCD_v2`.

Rules 3 and 4 update: Map the rescue day and place numbers to a rescue day and place, respectively, as follows below. Unmapped day or place numbers indicate an invalid phone number. Use nested `if-else` or `if-elif-else` selection, not lists or NumPy arrays. If your Python installation supports it, consider `match-case-other` selection. Decide which permitted approach yields more readable code.

	1	2	3	4	5	6	7
Rescue on	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
at the	village bridge	village library	village church	medical centre	bus terminal	railway station	nearest airport

If the inputted number passes all Rules, print a message to the Console indicating the rescue day and place on one line, as shown in a test case. For an invalid number, print one of the following messages only to indicate the *first* broken Rule only, meaning the broken Rule with the *lowest* Rule number: "`Decoy number: Not eleven digits`"; "`Decoy number: A digit sum is odd`"; "`Decoy number: Invalid rescue day`"; or "`Decoy number: Invalid rescue place`".

Divide the `RULES 1 TO 4` section meaningfully into four sections: `RULES 1 TO 2`, `RULE 3`, `RULE 4`, and `PRINT OUTPUT`. Produce all Console output in the `PRINT OUTPUT` section. Consider whether a few variables produced or modified in one section, to influence a subsequent section, would help to reduce or eliminate the nesting of selection statements inside other selection statements. When you write code in a more readable way, it helps to verify by reading/testing that it meets all/most requirements.

Do not use repetition (`for`/`while` loops) and/or invocation (user-defined function) program flows. Also, do not use `exit` statements of any kind. Execution begins, syntax errors aside, at the first non-blank non-comment line and must end, runtime errors aside, at the last non-blank non-comment line. Given test cases are a small subset of possible test cases. Compose and test additional cases yourself.

Ensure that every script section has a comment header that, in your own words, summarizes what the section does or is supposed to do. Review and revise, as appropriate, the program's initial header, attending to names of and percentages from others. Submit your Version 2 solution, `decode` only, by the Version 2 deadline. Rename it, as may be required by the lab instructor, before submission.

Revision History

This Python lab derives from a C/C++ one created by Paul Iglinski, pre-2010, and MATLAB derivatives by Sarah McEvoy, Edward Tiong, and Wing Hoy, 2011 to 2021. These instructions and related files were created and reviewed, in 2022, by Wing and Dileepan Joseph. They were reviewed and revised, in 2024, by Antonio Andara Lara, Wing, and Dr. Joseph. Encrypted Messaging Service, a Programming Contest entry, is an example of student work on secret messages that does not interfere with this lab.