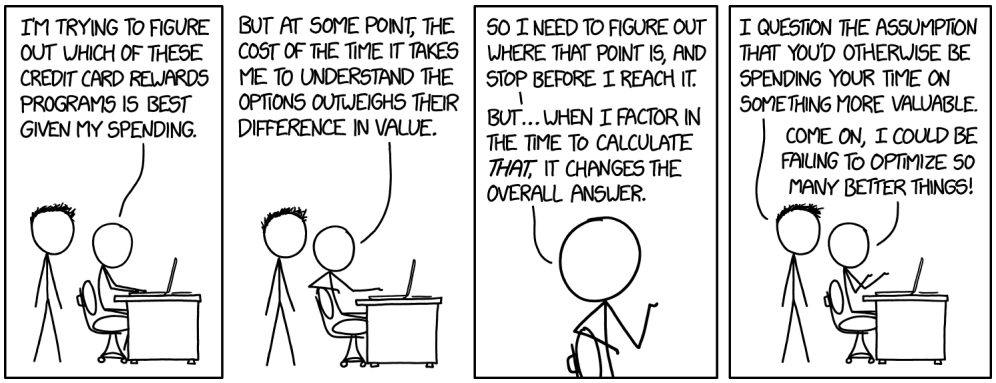
CS50 - Assignment 3
Credit Cards

Due: Monday, September 21 at 11:59am



`https://xkcd.com/1908/`

For this assignment we're going to be playing with generating and validating credit card numbers. A number of the functions utilize the math operators (`%`, `//`) to manipulate integers, so make sure you review those operators before you start.

Move the `assignment3` folder containing this pdf and `group.py` into your `Documents/cs50` folder, then open it in VS Code (you can use File→Open Folder).

(Note: you can download it again from: `https://cs.pomona.edu/classes/cs50/assignment3.zip`)

Loop Problems *(Group)*

Open up `group.py`. There are a number of functions in this file, and they're all wrong in some kind of way. In your LC, go through these problems one at a time. You can work individually or in pairs, but should check in with the rest of the group for each problem as you go. Repair the functions in `group.py` and double-check them in the interpreter. Note that some of them are only out of order or wrongly indented, while others are broken in substantial ways and will need complete rewriting.

Helper functions *(Individual)*

In VS Code, create a file called `assign3.py`.

Before we tackle the main functions, we're going to develop some helper functions that we can then use to help us on the credit card problems.

1. Write a function called `last_digit` that takes a positive integer as input and returns the last digit of that integer (as an `int`). Hint: review the math operators and think about how you can isolate that last digit.

   This must be done mathematically, and not by converting the number to a string. We haven't talked about strings yet!

2. Write a function called `integer_right_shift` that takes an integer as input and returns the integer constructed by removing the last digit. For example:

   ```
   >>> integer_right_shift(123)
   12
   >>> integer_right_shift(1)
   0
   ```

   Again, this must be done mathematically.

3. Write a function called `add_right_digit` that takes as input two positive integers. The first can be any positive integer and the second must be a single digit integer (i.e., 0-9). The function should return a new integer constructed from appending the single digit integer onto the *right* of the first integer. For example:

   ```
   >>> add_right_digit(1234, 5)
   12345
   >>> add_right_digit(474, 7)
   4747
   >>> add_right_digit(47,0)
   470
   ```

   This must be done mathematically. Think about what this involves.

4. Write a function called `sum_digits` takes an integer as input and returns the (integer) sum of the digits. You *must* use your previous two functions to help you write this function. It should work for numbers of any number of digits, so think about what tools Python provides for the situation where you don't know how many steps you'll need to perform in advance.

Testing the helper functions *(Individual)*

An important part of programming is convincing yourself that your functions are correct (In fact, this can sometimes be harder than writing the function itself!). In VS Code, create a new file called `assign3_tester.py`. At the top of this file add `from assign3 import *` to import all of your functions from your `assign3.py`.

Python has a statement called `assert` that should be followed by a `bool`. If the `bool` is `True` then nothing happens. If the `bool` is `False`, then it raises an error. You can try this out in the Python interpreter:

```
>>> x = 10
>>> assert x > 0
>>> assert x == 10
>>> assert x > 100
Traceback (most recent call last):
  File "<python-input-3>", line 1, in <module>
    assert x > 100
           ^^^^^^^
AssertionError
```

Notice in the first two cases where it's `True`, Python doesn't do anything. However, in the third case where it's `False`, it raises an error. We can use the `assert` statement to test our code.

5. In your `assign3_tester.py`, write a function called `test_last_digit` that takes no parameters and is of the following format:

   ```
   def test_last_digit():
       print("Testing last_digit")
       assert last_digit(47) == 7
       assert ...
       assert ...
       assert ...
       print("*** all tests passed ***")
   ```

   Your function should have at four asserts and you should try and come up with test different cases.

   When you're all done, you should be able to run your test function and, assuming `last_digit` is correct, should pass:

   ```
   >>> test_last_digit()
   Testing last_digit
   *** all tests passed ***
   ```

6. In your `assign3_tester.py`, write a function called `test_integer_right_shift` that follows the same outline as the previous function and includes at least four `asserts`.

7. In your `assign3_tester.py`, write a function called `test_add_right_digit` that follows the same outline and includes at least four `asserts`.

8. In your `assign3_tester.py`, write a function called `test_sum_digits` that follows the same outline and includes at least four `asserts`.

Credit Card Numbers

What makes a credit card number valid? For our purposes, a credit card number is valid if it passes the Luhn algorithm[1]. (In practice there are also additional restrictions. For example, all Visa cards must start with the number 4.)

An algorithm is a set of instructions for accomplishing a specific task, so the Luhn algorithm is a set of instructions for checking whether a number is potentially a valid credit card number.

Here are the main steps of the Luhn algorithm, using the number `9813428854407` as a worked example:

1. Double the value of alternate digits of the credit card number beginning with the second digit from the right (the first right-hand digit is the check digit).

   ```
   Original: 9 8  1 3 4 2 8  8 5 4 4 0 7
   Doubled:  9 16 1 6 4 4 8 16 5 8 4 0 7
   ```

2. Add all of the individual digits from the previous step:

   ```
   9 + (1+6) + 1 + 6 + 4 + 4 + 8 + (1+6) + 5 + 8 + 4 + 0 + 7 = 70
   ```

   We've added parenthesis to make it explicit where each of the digits come from, but it's just adding all of the digits of all of the numbers.

3. If the total obtained in the previous step is a number ending in zero (30, 40, 50, etc.), then the account number is valid.

   70 ends in a 0, so the account number is valid.

Before you continue, please make sure you fully understand the Luhn algorithm and how it checks numbers for validity! It doesn't make sense to try to implement an algorithm until you understand how it works. Please feel free to discuss the algorithm with other students, the TAs, the professors, etc! Make sure you understand it well enough to answer the following questions by hand (a calculator is fine to help with the sums):

[1] `https://en.wikipedia.org/wiki/Luhn_algorithm`

Is 1234567890123 a valid number?

If the first 12 digits of a 13-digit credit card number are 111111111111, what must the 13th digit be?

Verifying Credit Card Numbers *(Individual)*

7. Write a function called `verify` that takes a 13 digit integer as input and returns `True` if the number passes the Luhn algorithm and `False` if it fails.

8. In your `assign3_tester.py` file, write a function `test_verify` that takes zero parameters, follows the format of the other testing functions, and has at least 4 `asserts`.

   There are websites that will calculate Luhn sums for you. You can use one of these to help come up with test cases, e.g., `https://planetcalc.com/2464/`. Don't use your actual credit card number as a test case!

Generating Credit Card Numbers *(Individual)*

9. Write a function called `generate` that takes a 6 digit integer as input (in the range 100000-999999) and returns a valid 13 digit credit card number which begins with the given 6 digits. The return value must be of type int.

   There are a couple of strategies for how to tackle this problem. Whatever you do should choose at least some of the digits randomly. Pick whichever approach makes the most sense to you (or try both, or come up with a third!).

   Strategy 1:

   - Add 6 additional random digits to the input 6-digit number.
   - Figure out what the last digit (the check digit) needs to be based on the current 12 digits. For example, if your first 12 digits are `123456789012` then the last digit needs to be an 8.

   Strategy 2:

   - Add 7 additional random digits to the 6-digit input number.
   - Check if it's a valid credit card number. If not, replace the *last* digit with a new random digit and check again. Repeat this until it's valid.

   You could even generate the 7 digit sequence randomly over and over again, stopping when it is finally valid. Ask yourself why we might prefer one of the approaches above.

10. In your `assign3_tester.py` file, write a function `test_verify` that takes zero parameters, follows the format of the other testing functions, and has at least 4 `asserts`. You can utilize your `verify` function to help you with this.

When you're done

Make sure that your program is properly commented:

- You should have comments at the very beginning of the file stating your name, course, assignment number and the date.
- Each function should have an appropriate docstring (see details in the Style section that follows on what this should include).
- Other miscellaneous comments to make things clear

In addition, make sure that you've used good *style*. This includes:

- Following naming conventions, e.g. all variables and functions should be lowercase.
- Using good variable names.
- Docstrings should include information on each parameter and its purpose, as well as what sort of value the function returns and what it signifies.

  You can use syntax like this to describe parameters and the return value:

```
def example_function(in1, in2):
    """
    This is an example function to show how to describe parameters in docstrings.

    :param in1: (int) the number of points per correct answer
    :param in2: (bool) whether this is a quiz or a midterm
    :return: (float) the mean score of student submissions
    """
    # ... function goes here ...
```

- Proper use of whitespace, including indenting and use of blank lines to separate chunks of code that belong together.

What to submit

For this assignment you should submit three things AT THE SAME TIME (see below):

1. Your `assign3.py` file. **Note:** The file you submit should only contain your function definitions. There shouldn't be any calls to those functions (except inside other function definitions) and there should be *no* calls to `print`. (See "Grading" below for the list of required functions.)
2. Your `assign3_tester.py` file.
3. Your `group.py` file.

Grading

	points
`last_digit`	1
`integer_right_shift`	1
`add_right_digit`	1
`sum_digits`	2
`verify`	3
`generate`	3
`test_last_digit`	1
`test_integer_right_shift`	1
`test_add_right_digit`	1
`test_sum_digits`	1
`test_verify`	1
`test_generate`	1
Comments, style	2
total	19