Com S 127x
Fall 2016
Assignment 4

Due Date: Wednesday, November 2, 11:59 pm (midnight)
"Late" deadline (25% penalty): Thursday, November 3, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus for details.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away.

Introduction

The purpose of this assignment is to give you some practice with loops and strings, procedural decomposition, and unit testing. It is also going to incidentally give us some practice with number systems (such as the binary number system used within your computer's processor).

We will be working in a variation of the number system used by the Maya civizilation, which flourished in the region that is now southern Mexico and Central America from 2000 BCE until about 400 years ago (when they were wiped out by the Spanish). The Maya are credited with having invented the concept of "zero", which makes possible the place value number systems we use today.

For our purposes a "Mayan number" will be a string of text. Details are given in the next section. Your task will be to create two modules:

`mayan.py` - various functions for working with Mayan numbers
`mayan_test.py` - a set of unit tests for the functions in mayan.py

Converting between number representations

This section contains some examples of how to convert between number representations. This topic is sometimes covered in middle school but we tend to forget these things, and as a computer scientist or engineer you'll need to know this stuff.

The number system we normally use is called a *place value* system because the meaning of a given digit depends on which "place" it's in. For example, the "4" in "42" has the value 40, but the "4" in "4096" has the value 4000. We usually write numbers in *decimal* or *base ten*, which just means that we use powers of 10 to interpret the values of the digits in a number:

$10^0 = 1$
$10^1 = 10$
$10^2 = 100$
$10^3 = 1000$
$10^4 = 10000$
...

Base 10 also means that we need 10 different symbols to represent the digits, including a zero. The value of a string of digits is given by adding up the values of the digits in each place, multiplied by the corresponding power of 10.

Place value	10^3	10^2	10^1	10^0
Digit	**4**	**0**	**9**	**6**

$\mathbf{4} * 10^3 + \mathbf{0} * 10^2 + \mathbf{9} * 10^1 + \mathbf{6} * 10^0 = 4000 + 0 + 90 + 6 = 4096$

Many of us also have to work frequently in *binary*, or *base 2*. It works the same way except the place values are given by powers of 2:

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
...
For base 2, we only need two digits, 0 and 1. For example, here is how we would interpret the string "1101101"

Place value	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Digit	**1**	**1**	**0**	**1**	**1**	**0**	**1**

$\mathbf{1} * 2^6 + \mathbf{1} * 2^5 + \mathbf{0} * 2^4 + \mathbf{1} * 2^3 + \mathbf{1} * 2^2 + \mathbf{0} * 2^1 + \mathbf{1} * 2^0 = 64 + 32 + 0 + 8 + 4 + 0 + 1 = 109$

To convert between representations is not hard. The calculation above suggests an algorithm for converting from some base *b* to base 10:

```
result = 0
power = 1
for each digit, working right to left
        result = result + digit * power
         power = power * b
```

There are several strategies for converting from base 10 to base *b*. The simplest is probably to just look at the remainders as you repeatedly divide by *b*. For our base 2 example,

1 = 109 % 2, 109 / 2 = 54
0 = 54 % 2, 54 / 2 = 27
1 = 27 % 2, 27 / 2 = 13
1 = 13 % 2, 13 / 2 = 6
0 = 6 % 2, 6 / 2 = 3
1 = 3 % 2, 3 / 2 = 1
1 = 1 % 2, 1 / 2 = 0

You can stop when you finally divide by *b* and get zero. In pseudocode,

```
while number > 0
        digit = number % b
        number = number / b
        add characters for digit to the output string
```

This strategy generates the values of the digits in right-to-left order (read the leftmost column bottom-to-top in the example). So if you're writing the characters to a string, either you'll have to concatenate on the left, or else reverse the string when you're done.

Mayan numbers

The Maya are credited with having invented the concept of *zero*, which is crucial to having a place value system. Their number system was essentially base 20[1]. When numbers were written, though, they did not use 20 distinct digits. Instead, each "digit" was either a zero, or was a combination of lines and dots. A line was value 5 and a dot was 1. There are at most three lines, followed by at most four dots, in the following combinations.

[1] Rather than using powers of 20, as we are doing here, the place values were 1, 20, 18 * 20, 18 * 20^2, 18 * 20^3, etc. It seems likely that the value for the third place, 20 * 18 = 360, was chosen because it was the number of days in a year in their calendar. For this assignment we'll use a pure base 20 system, however

0	0
1	*
2	**
3	***
4	****
5	|
6	|*
7	|**
8	|***
9	|****

10	||
11	||*
12	||**
13	||***
14	||****
15	|||
16	|||*
17	|||**
18	|||***
19	|||****

The Maya actually wrote numbers vertically, but we'll use the convention shown above, using the characters "|" (the vertical bar above the enter key) and "*". In addition, we'll always put at least one whitespace character between each group of characters making a "digit". For example, to interpret the following string:

```
|||* 0 ||** ***
```

We could first interpret the individual "digit groups" as numbers 0 through 19:

Place value	20^3	20^2	20^1	20^0
Digit	**16**	**0**	**|12**	**3**

16 * 20^3 + **0** * 20^2 + **12** * 20^1 + **3** * 20^0 = 16 * 8000 + 0 * 400 + 12 * 20 + 3 * 1
= 128000 + 0 + 240 + 3 = 128243

Detailed specification for `mayan.py`

Your basic task is to write the following functions that will convert from Mayan numbers to integers and from integers to Mayan numbers. As usual, we'll use procedural decomposition to break the general problem down into simpler functions:

`convert_to_digit_group(n)`

Given an integer n between 0 and 19, inclusive, returns a string representing the corresponding Mayan number. If n is not between 0 and 19 the function returns an empty string.

`convert_to_mayan_number(n)`

Given any non-negative integer n, returns a string representing the corresponding Mayan number with exactly one space character between digit groups and no leading/trailing whitespace. If n is negative the function returns an empty stsring.

`convert_from_digit_group(m)`

Given a string m representing a valid Mayan digit group, returns the integer value. If m is not a valid Mayan digit group, returns -1. Extra leading or trailing whitespace is not allowed.

`convert_from_mayan_number(m)`

Given a string m representing a valid Mayan number, returns the integer value. If m is not a valid Mayan number, returns -1. Extra whitespace is allowed.

`is_valid_digit_group(m)`

Given a string m, determines whether the string is a valid Mayan digit group. That is, the function returns True if the string is the single character `"0"` or contains 0, 1, 2, or 3 `"|"` characters followed by 0, 1, 2, 3, or 4 `"*"` characters, and returns False otherwise. Extra leading or trailing whitespace is not allowed.

Writing test cases

In the last assignment we worked on a problem using turtles. Turtles are nice because you get some visual feedback, but in general, graphical code is actually quite difficult to test. This project is simpler, because it is easier to test.

To get your started, we have provided a sample module **`mayan_test.py`**. The file includes test cases for **`convert_to_digit_group`**. At the top of the file, notice there are five "helper" functions to use when writing tests. These functions save some of the tedious work of checking values and printing error messages.

Your task is to add test cases for the remaining four functions. The goal of your tests is to reveal bugs in any implementation of the specified function (not just your own). You want to be able to confidently assert to your boss that if someone's **`mayan.py`** module passes all your tests, then it is correct.

We will evaluate your test code by running it on some implementations with and without bugs. If there is an error in the code, at least one of your test cases should fail. Each test case should include a comment stating what you are intending to test and should use the corresponding helper function. See the examples in **`mayan_test.py`**.

Getting started

Remember our basic problem solving principles:

- Do concrete examples first.
- Can I solve part of the problem?
- Can I solve a related, simpler problem?

1. You already have test cases for **`convert_to_digit_group`**. You might start with that function. Given a number like 17, what is its Mayan representation? How many fives are there? How ones are left over? What should I do if the number is outside the range, like -1 or 20?

2. If you have **`convert_to_digit_group`** working, try **`convert_to_mayan_number`**. First write a few concrete test cases. For example, what's the Mayan representation of 137? How about 763?

3. What's a related, simpler problem? How about this: given an integer, can you determine just integer value of the *last* Mayan digit group? As an example, given 137, how do you know the last digit group will be 17, or **`"|||**"`**? What would be the next digit group to the left? Can you implement the algorithm on page 3 so that it just *prints* the values for each Mayan digit? (For example, given 128243, print out the values 3, 12, 0, 16?)

4. Likewise, for **`convert_from_digit_group`**, write a few test cases. Start by assuming that the string is valid. (Later, modify the function to rely on **`is_valid_digit_group`** to check whether the string is valid.) What's the value of **`"||**"`** ? How about **`"|****"`** ? You should be able to just add 5 for each **`"|"`** character and 1 for each **`"*"`** character. and return the total.

5. If you have **`convert_from_digit_group`** working, how about a number with two Mayan digit groups, like "** ***" ? How about three, for example, "* ** ***" ? Do the examples by hand so that you know what you're looking for. Then, remember that in Python, the string **`split()`** method is the easiest way to split apart the digit groups so you can get their values one at a time. Try just printing out the value of each group, for example, given "* ** ***", can you write a loop that will just print out the numbers

```
1
2
3
```

Then, can you do it in reverse? For example, you'd get the output

```
3
2
1
```

Then, can you implement the algorithm on p. 3 to add up the correct powers of 20?

6. Now think about **`is_valid_digit_group.`** (This may actually be the hardest function to write.) Give some examples of valid and invalid digit groups. How do you determine what's valid and what's not? Is **`"**0"`** valid? How about **`"*||*"`** ? Go ahead and write some test cases in **`mayan_test.py`**. Now modify your **`convert_from_digit_group`** so it returns -1 if the digit group is invalid.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`hw4`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw4`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Python examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it.

Documentation and style

Include comments at the top of each file with

- your name, and
- a brief statement of what the program does

For each *function* you write, include a comment that states what it does. For functions, Python programmers prefer to put such comments in the form of a "docstring". A docstring is just a string of text, appearing directly after the function's **`def`** line, surrounded by triple quotes. See the sample code **`mayan_test.py`** for examples.

What to turn in

Turn the two files **`mayan.py`** and **`mayan_test.py`**.
Upload your two files to the Homework 2 submission link on Blackboard. Be sure to CHECK your submission history and make sure that you successfully submitted both files. **Remember you can't just upload the files, you have to click the "Submit" button!**