Problem Set 4

Handed out: Sept 28, 2017

Due: Oct 6, 2017 @ 5:00 PM

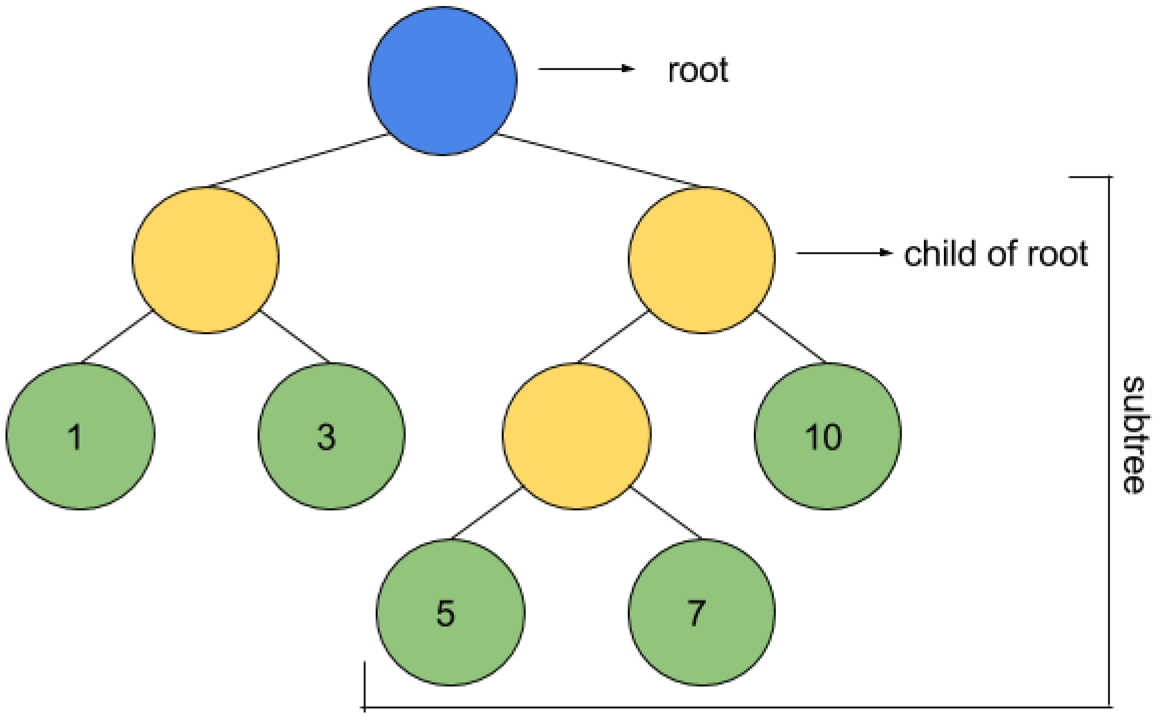
This problem set has two parts: A and B. They are not dependent on one another, so feel free to work on them in parallel, or out-of-order.

Do not rename the files we provide you with, change any of the provided helper functions, change function/method names, or delete provided docstrings. You will need to keep **`trees.pyc`** in the same folder as **`ps4a.py`** and **`words.txt`** and **`story.txt`** in the same folder in which you store **`ps4b.py`**.

Finally, please consult the Style Guide on Stellar, as we will be taking point deductions for violations (e.g. non-descriptive variable names and uncommented code). For this pset style guide numbers **6, 7 and 8** will be highly relevant so make sure you go over those before starting the pset, and again before you hand it in!

Part A : Recursive Operations on Trees

A tree is a hierarchical data structure composed of linked nodes. The highest node is called the *root*, which has *branches* that link it to other nodes, which are themselves roots of their respective *subtrees*. A simple tree is shown below:



In this tree, the **root** is indicated by **blue**, the **intermediate nodes** (nodes that themselves have children) are indicated by **yellow**, and **leaves** (nodes without children) are indicated by **green**.

We can make a few observations for this specific tree:

- Leaves hold data: in this case, integers. We can also consider trees where branches also hold data, but we will restrict ourselves to leaves this time.
- Data is hierarchical: each node has a **parent** (except the root) and each node has **1 or more children** (except the leaves). We will be using this nomenclature in the rest of the problem set.
- Trees are inherently recursive: the right branch of the root (a **subtree**) is itself a tree.

Part A0: Data representation practice

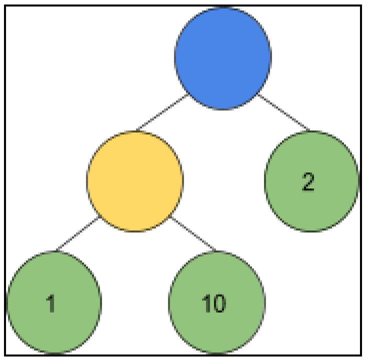
In this problem set, we will be using lists to represent trees. Since lists do not have hierarchical information themselves, we will be using sublists to indicate subtrees.

The tree above can be represented as the following list:

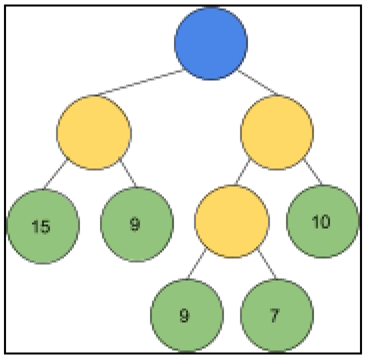
```
example_tree = [[1,3],[[5,7],10]]
```

We represent each leaf with the integer value it holds, and non-leaves (root or intermediate nodes) with a list. Note that the recursion property holds: if we consider the right branch, `[[5,7],10]`, we see that it is also a valid tree.

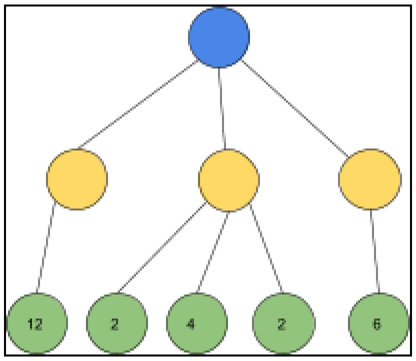
We will practice this notation in this part. For the trees shown below, create lists accurately representing the data. Put them into the variables at the top of ps4a.py, named **tree1**, **tree2** and **tree3**.



Tree 1



Tree 2



Tree 3

When you correctly initialize the three variables, the test named **`test_data_representation`** in test_ps4a.py should pass.

Part A1: Multiplication on tree leaves

Write a recursive function, **`mul_tree`**, that multiplies all the values of the leaves in each tree. **This function must be recursive; non-recursive implementations will receive a zero.**

Hint: The following pseudocode may prove to be helpful

```
Given an input tree, T:
Base case: If T is an empty list [], the product of its leaves would be 1.

Recursive cases:
Case 1: The first child of T is a leaf. We could then get the product of the
entire tree by multiplying this leaf's value to the product of the leaves of
all of its siblings.

   Example:
   T = [2,[2,5],[3,4]]]
   The product of T is the value of its first child (2) multiplied by the
   product of [[2,5],[3,4]] (which is just the original tree with the first
   child removed.)

Case 2: The first child of T is also a tree. We could then get the product of
the entire tree by multiplying the product of the leaves of that child to the
product of the leaves of its siblings.

   Example:
   T = [[1,2],[3,[4,5]]]
   The product of T is the product of the child tree [1,2] times the product of
   the tree [[3,[4,5]]] (which again, is just the original tree with the first
   child removed.)
```

You should test your function using the variables from the previous part. For example:

```
mul_tree(tree1) # should be 20
mul_tree(tree2) # should be 85050
mul_tree(tree3) # should be 1152
```

Now, your code should also pass the test **`test_mul_example_trees`**.

Part A2: Arbitrary operations on tree leaves

Implement **`operate_tree`**, that can carry out arbitrary operations on the leaves of a tree. This function also has to be recursive, and it should take in three parameters: the tree **`tree`**, the operation to execute **`op`**, and the value that the function should return in base case (where the tree is empty), **`base_case.`**

Example operations (**`op)`** are given below:

```
def addem(a, b):                def prod(a, b):
     return a + b                    return a * b
```

Example usage below. After you write your function, you should test it by printing out the results of these function calls:

```
operate_tree(tree1, addem, 0) # should be 13
operate_tree(tree1, prod, 1)  # should be 20
operate_tree(tree2, addem, 0) # should be 50
operate_tree(tree2, prod, 1)  # should be 85050
```

Hints:

+ **`op`** will be a function by itself --- not the result of a function call.

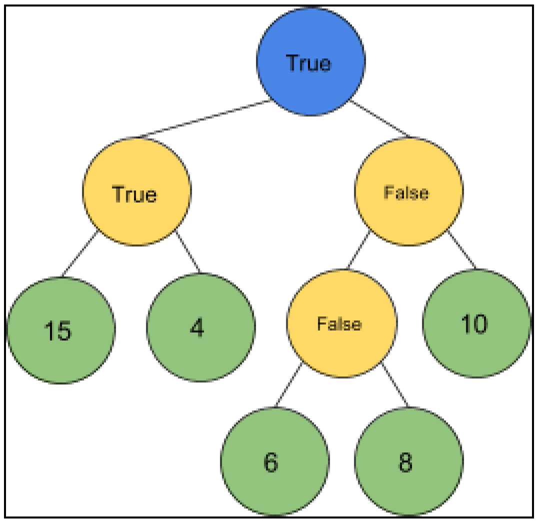
+ You should be able to reuse the structure of your **`mul_tree`** function, only changing a few lines of code. Think about the places where you use multiplication in the previous function, and how you could change that function to use "**`prod`**" instead of "**`*`**". Then, think about how to generalize that to work with with any **`op`** and any **`base_case`** value.

You should now pass **`test_op_add_example_trees`** and **`test_op_prod_example_trees`**.

Part A3: Searching a tree

Now we will try writing our own operation function, similar to **`prod`** and **`addem`** from the last section. This time, the function is called **`search_odd`**. It should return **`True`** if one of the tree's leaves is odd, and **`False`** otherwise.

This operator will be slightly more complicated that **`addem`** and **`prod`** were, because it needs to be able to handle both booleans *and* integers as input. Consider the following tree:



At the non-leaf nodes, you can see the True/False decision, and therefore what the function **`search_odd`** should return, for each sub-tree. Looking at the left two yellow nodes, we see that **`search_odd(15, 4)`** should return True, while **`search_odd(6, 8)`** should return False.
However, we aren't always combining two numbers -- note that at the rightmost child of the root, we need to combine a boolean from the lower subtree with a number, 10. In that case, we'd be calling **`search_odd(False, 10).`**

Therefore, your function **`search_odd(a, b)`** should behave as follows:

- If **`a`** and **`b`** are both integers, return **`True`** if *either of them* are **`odd`**.
- If **`a`** and **`b`** are both booleans, return **`True`** if *either of them* are **`True`**.
- If one of **`a`** and **`b`** is an integer, and the other is a boolean, return **`True`** if the boolean is **`True`** or the integer is **`odd`** (or both).

You should now pass all of the tests in **test_ps4a.py**.

Part B: Cipher Like Caesar

Introduction

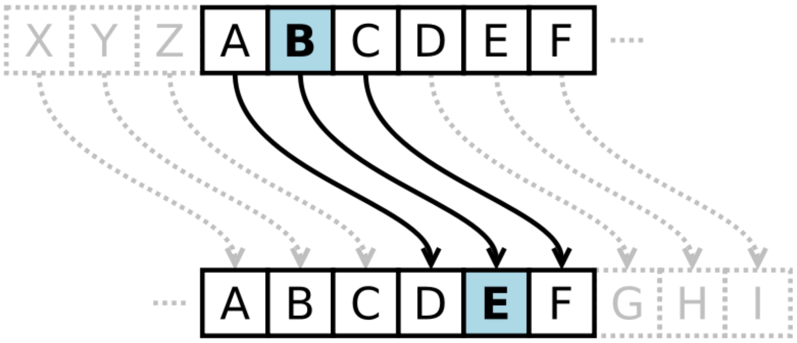
In this problem we will implement a simple cipher.

Here is some important vocabulary we will use going forward:

- **Encryption** - the process of obscuring or encoding messages to make them unreadable
- **Decryption** - converting encrypted messages back to their original, readable form
- **Cipher** - a technique or algorithm used for performing encryption and decryption
- **Plaintext** - the original, readable message
- **Ciphertext** - the encrypted message. *A ciphertext still contains all of the original message information, even though it looks like gibberish.*

How the Caesar Cipher Works

The idea of the Caesar Cipher (Wikipedia page here) is to pick an integer amount and shift every letter of your message by that amount.
Suppose the shift we choose is `k`. Then, all instances of the ith letter of the alphabet that appear in the plaintext should become the (i + k)th letter of the alphabet in the ciphertext.
Consider the case where we use `k = 3`.This results in a mapping of letters as follows:



We will be implementing a **modified** version of the Caesar Cipher. Instead of selecting one integer, k, to shift all letters by, we will be using a **list of integers**. You will iterate through the list of integers in order to determine the shift value, looping around the list as needed. For the purposes of this homework, **you can assume that this list will always be of length 2**. Even when you come across characters that do not change, you should still iterate to the next shift value in the list. See examples in table below.

Example:

```
shifts = [1, 3]

plaintext = “donut”

# d + 1 = e, o + 3 = r, n + 1 = o, u + 3 = x  , t + 1 = u

ciphertext = “eroxi”
```

You will need to be careful to properly handle the case where the shift “wraps around” to the start of the alphabet, when `i+k > 26` (such as with X, Y, and Z in the diagram above.)

We will treat uppercase and lowercase letters separately, so that uppercase letters are always mapped to an uppercase letter, and lowercase letters are always mapped to a lowercase letter. If an uppercase letter maps to “A”, then the same lowercase letter should map to “a”.
Punctuation, spaces, and numbers should stay the same as they were in the original message. For example, a plaintext message with a comma should have a corresponding ciphertext with a comma at the same position.

Plaintext	**Shift values**	**Ciphertext**
“aaaaaaaa”	[2,5]	“cfcfcfcf”
“abcdef”	[2, 5]	“cgeigk”
“Hello, World!”	[4, 0]	“Lepls, Wsrpd!”
“I am 19 years old”	[3, 1]	“L dn 19 bfdsv rmg”
“......”	Any values	“......”
“” (empty string)	Any values	“”

Using Classes and Inheritance

This is your first experience coding with classes! Get excited! They are a powerful idea that you will use throughout the rest of your programming career.

For this problem set, we will use a parent class called **`Message`**, which has two child classes: **`PlaintextMessage`** and **`CipherTextMessage`**.

- **`Message`** contains methods that both plaintext and ciphertext messages will need to use -- for example, a method to get the text of the message. The child classes will inherit these shared methods from their parent.
- **`PlaintextMessage`** contains methods that are specific to a plaintext message, such as a method for encrypting a message given its shift value k.
- **`CiphertextMessage`** contains methods that are specific to a ciphertext, such as a method to decrypt a message.

There are a few helper functions we have implemented for you: **`load_words`**, **`is_word`**, and **`get_story_string`**. You may use these in your solution.
You don't need to understand exactly how they work, but you should read the associated comments to understand what they do, and how to use them.

Part 1: Message

Fill in the methods of the Message class found in ps4b.py, according to the specifications in the docstrings. Please see the docstring comment with each function for more information about the function's specification.

We have provided skeleton code in the Message class for the following functions. Your task is to fill them in.

- `__init__(self, text)`
- `get_message_text(self)`

This should return an immutable version of the message text we added to this object in init. Luckily, strings are already immutable objects, so we can simply return that string.

- `get_valid_words(self)`

This should return a COPY of self.valid_words to prevent someone from accidentally mutating the original list.

- `build_shift_dicts(self, shifts)`

You may find the string module's ascii_lowercase and ascii_uppercase constants helpful. Additionally, keep in mind the modulo operator, `%`, which returns the remainder when division is performed: for example,

```
>>> 10 % 8
2
>>> 5 % 7
5
>>> 2 % 2
0
```

- `apply_shifts(self, shift_dicts)`

Remember that spaces, punctuation and numbers should not be changed when the cipher is applied.

You can test out your code so far by running test_ps4b.py. Make sure it's in the same folder as your problem set. At this point, your code should be able to pass all the tests beginning with "`test_message`", but not the ones beginning with "`test_plaintext_message`" or "`test_ciphertext_message`" -- we will implement code for those in the next section.

Part 2: PlaintextMessage

Fill in the methods of the PlaintextMessage class found in ps4b.py according to the specifications in the docstrings.

- `__init__(self, text, shift)`

You should use the parent class constructor in this method to make your code more concise. Take a look at Style Guide #7 if you are confused.

- `get_shifts(self)`
- `get_encryption_dicts(self)`

Note: this should return a COPY of self.encryption_dicts to prevent someone from mutating the original dictionary.

- `get_message_text_encrypted(self)`
- `change_shifts(self, shifts)`

Think about what other methods you can use to make this easier. It shouldn't take more than a couple lines of code.

You can test your new class by running test_ps4b.py. You should now be able to pass all the tests in that file, **except** for `test_ciphertext_decrypt_message`.

Part 3: CiphertextMessage

Given an encrypted message, if you know the shift used to encode the message, decoding it is trivial. If `message` is the encrypted message, and `[a, b]` are the shifts used to encrypt the message, then `apply_shift(message, [26-a, 26-b])` gives you the original plaintext message.

The problem is that you won't be given the shifts along with the encrypted message. But fortunately, our encryption method only has 26^2 distinct possible values for the shift list. Since we know the messages are written in English, if we write a program that tries each shift and maximizes the number of valid English words in the decoded message, we can decrypt the ciphertext.

Fill in the methods of the CiphertextMessage class found in ps4b.py according to the specifications in the docstrings.

- `__init__(self, text)`

As with PlaintextMessage, use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.

- `decrypt_message(self)`

You may find the helper function `is_word(wordlist, word)` and the string method `split` useful. Note: `is_word` will ignore punctuation and other special characters when considering whether a word is valid.

You should now be able to pass **all** the tests in test_ps4b.py.

Part 4: Writing Your Own Tests

Write two test cases for `PlaintextMessage` and two test cases for `CiphertextMessage` in the functions `test_plaintext_message()` and `test_ciphertext_message()`, at the bottom of ps4b.py. The function comments contain some sample tests to get you started.

Each test case should display the **input**, **expected output**, and **actual output**. Each case should handle a different case for the inputs: for example, you might write one to test that messages with capital letters are correctly encoded. Put a comment above each test case you write explaining what it is testing. *Remember the importance of using getters and setter functions!*

Run your test cases by uncommenting the specified lines at the bottom of the file under `if __name__ == '__main__'`.

Now, try out using your classes to decode the file story.txt. Write the code to do this inside the function `decode_story()`.

Hint: The skeleton code contains a helper function `get_story_string` that returns the encrypted version of the story as a string. Create a `CiphertextMessage` object using the story string, and then use `decrypt_message` to return the appropriate shift value and unencrypted story.

You test will be graded during checkoff so be prepared to run them for us.

Hand-In Procedure

1. Save

Save your solutions with the original file names: **ps4a.py, ps4b.py**. *Do not ignore this step, or save your file(s) with a different name!* Make sure your code runs without throwing syntax errors, and passes all of the provided test cases in test_ps4a.py, and test_ps4b.py. Make sure to delete any debugging print statements or commented-out sections of code.

2. Time and Collaboration Info

At the start of each file, fill out the comment section with the number of hours (roughly) you spent on the problems in that part, the names of the people you collaborated with, and number of late days you used, if any. For example:

```
# Problem Set 4B
# Name: Ana Bell
# Collaborators: John Guttag, Eric Grimson
# Time Spent: 8:30
# Late Days Used: 1
```

3. Submit

To submit a file, upload it to the section for Problem Set 4 on the submission portal. You may upload new versions of each file until the **11:59pm** deadline, but anything uploaded after that time will be counted towards your late days, if you have any remaining. If you have no remaining late days, you will receive no credit for a late submission.