Problem Set 4: Recursion and Caesar Cipher

Pset Buddy

You do not have a buddy assigned for this pset.

Introduction

This problem set is split into two topics: the first focuses on recursion and trees (Part A), while the second looks at classes and cryptography (Parts B and C). The two topics are not dependent on one another, so feel free to work on them in parallel, or out-of-order. Please carefully read through the instruction for each.

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 words.txt, story.txt and pads.txt in the same folder in which you store your .py files.

Finally, please consult the Style Guide, 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!

1) 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:

![Tree Diagram]

We can make a few observations:
- Each node can hold data: in this example they are holding a string with their node type.
- Data is hierarchical: each node has a parent (except the root), each non-leaf node has 1 or more children, and each leaf node has 0 children. We will be using this nomenclature in the rest of the problem set.
- Trees are inherently recursive: a root's children nodes are "roots" of other smaller trees (called subtrees).

### 1.1) Data Representation Practice

In this problem set, we will be using a provided Node object in `tree.py` to represent trees.

```
example_tree = Node(1, Node(2), Node(5, Node(7), Node(8)))
```

The simple tree above can be initialized with the Node object as follows:

A brief explanation of the Node class is below:

- You can initialize a node with the following `Node(value, left_child, right_child). value holds the value held in the node, left_child optionally holds the Node constructing the left subtree, right_child does the same for the right subtree. If there is not a subtree, either do not input that parameter or pass in None.
- You can get the `Node` object holding the left or right subtrees with `get_left_child()` or `get_right_child()` respectively. If there is no child, this function returns None.
- You can get the value held by a `Node` with `get_value()`.

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

```
Tree 1
1
  2
   3
  4

Tree 2
1
  2
   3
  4

Tree 3
5
  6
   7
  8
```

### 1.1.1) Testing
1.2) Determining The Height of a Tree

The height of a tree is the number of edges between the root and the furthest leaf. For example, in the trees you initialized above tree 1 has a height of 2 and trees 2 and 3 have a height of 3. Write a recursive function, find_tree_height, that determines the depth of a tree. This function must be recursive; non-recursive implementations will receive a zero.

**Hint.** The following approach may prove to be helpful:

- Given an input tree, T:
  - **Base case.** If T is a leaf, it has a height of 0.
  - **Recursive case.** Recursively find the height of T's left and right subtrees and take the maximum of the two. Add 1 to the maximum height and return that value.

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

```python
find_tree_height(tree1) # should be 2
find_tree_height(tree2) # should be 3
find_tree_height(tree3) # should be 3
```

1.2.1) Testing

Now, your code should also pass the tests test_tree_height and tree_height_additional in test_ps4a_student.py.

1.3) Heaps

A special type of trees are **heaps**. There are two types of heaps: **max heaps** and **min heaps**.

In a **max heap**, for each node N, N is the maximum value in the tree rooted at N. This means that all of the elements stored in N's left and right subtrees are less than than the value stored in N.

Conversely, in a **min heap**, N is the minimum value in the tree rooted at N, so all of the elements stored in N's left and right subtrees are greater than the value stored in N.

![Max Heap](image1.png)

![Min Heap](image2.png)

**Write the function is_heap to quickly determine if a tree is a max or min heap, depending on the compare_func parameter.** The compare_func is a function that takes in two arguments, child_value and parent_value. For max heaps, this function will return True if child_value < parent_value and False otherwise. For min heaps, the function will return True if child_value > parent_value and False otherwise. **Conceptually, this allows you to write one function that can determine both max and min heaps based on a parameter, instead of writing two separate methods with very similar code.** Below, we have provided you with working implementations of compare_func for a max and min heap, respectively.
# max heap comparator
def compare_func(child_value, parent_value):
    if child_value < parent_value:
        return True
    return False

# min heap comparator
def compare_func(child_value, parent_value):
    if child_value > parent_value:
        return True
    return False

**Hint.** The following approach may prove to be helpful:

- **Given an input tree, T:**
  - **Base case.** If T is a leaf then it is a heap (for both max and min heaps)
  - **Recursive case.** For a node T, recursively check if T’s right and left subtrees are also heaps and that T is the max or min element (using the comparator function compare_func passed to is_heap) of its subtree. If all the previous conditions are true, then the tree rooted at T is a heap. Otherwise, it is not.

1.3.1) Testing

You should now pass all of the tests in test_ps4a_student.py.

2) Part B: Encryption with One Time Pads

2.1) Introduction

In this problem we will implement a simple encryption technique that when implemented correctly cannot be broken.

Here is some important vocabulary we will use going forward:

- **encryption.** the process of obscuring or encoding messages to make them unreadable.
- **decryption.** the process of converting encrypted messages back to their original, readable form.
- **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.

2.1.1) How a One Time Pad Works

The idea of a **one time pad** is to "shift" each character in your plaintext message by a random amount. This results in a ciphertext that cannot be decrypted without the list of random amounts your letters were shifted by, called the **pad**.

In this psset we want our messages to be able to include letters, numbers, spaces and other special characters. Shifting a letter makes sense (for example a shifted by 3 would be d), but what would a space shifted by 3 be? Luckily, all characters are represented on our computers as numbers already! You can view the mappings of basic characters to numbers in this ASCII table. In it we can see that a space is represented by the numeric value 32, so shifting it by 3 results in 35 or the character #. An example of shifting letters by 3 is pictured below.
We need to be careful to properly handle the case where the shift wraps around to the start. We’re only interested in characters with ASCII values from 32 to 126. So the last character ~ with the value 126 shifted by 1 would be a space (value 32) and ~ shifted by 5 would be a $ (value 36). Some more examples are found in the following table:

<table>
<thead>
<tr>
<th>Character (&amp; ASCII Value)</th>
<th>Shift value</th>
<th>Shifted Character (&amp; ASCII Value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot; (65)</td>
<td>5</td>
<td>&quot;F&quot; (70)</td>
</tr>
<tr>
<td>&quot;a&quot; (97)</td>
<td>10</td>
<td>&quot;k&quot; (107)</td>
</tr>
<tr>
<td>&quot; &quot; (SPACE, 32)</td>
<td>-1</td>
<td>&quot;.&quot; (126)</td>
</tr>
<tr>
<td>&quot;.&quot; (45)</td>
<td>8</td>
<td>&quot;.&quot; (53)</td>
</tr>
<tr>
<td>&quot;:&quot; (125)</td>
<td>4</td>
<td>&quot;:&quot; (QUOTE, 34)</td>
</tr>
<tr>
<td>&quot;k&quot; (107)</td>
<td>-3</td>
<td>&quot;h&quot; (104)</td>
</tr>
<tr>
<td>&quot;y&quot; (89)</td>
<td>-12</td>
<td>&quot;M&quot; (77)</td>
</tr>
<tr>
<td>&quot;Y&quot; (89)</td>
<td>-107</td>
<td>&quot;M&quot; (77)</td>
</tr>
</tbody>
</table>

Now that we know how to shift characters, to use a one time pad we simply have to shift each character in our message by a random amount specified by the pad. For example, if we have the message hello (ASCII values [104, 101, 108, 108, 111]) and the pad [3, 0, 10, 11, 4], we would do compute [104+3, 101+0, 108+10, 108+11, 111+4], which equals [107, 101, 118, 119, 115]. Converting back from values to characters, our ciphertext would be kevws. More examples are in the chart below:

<table>
<thead>
<tr>
<th>Plaintext</th>
<th>One Time Pad</th>
<th>Process</th>
<th>Ciphertext</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>1,2,3</td>
<td>65+1=66, 97+2=99, 97+3=100</td>
<td>Bcd</td>
</tr>
<tr>
<td>XYZ</td>
<td>2,10,12</td>
<td>120+2=122, 121+10=131, 122+12=135</td>
<td>z$'</td>
</tr>
<tr>
<td>Hello!</td>
<td>5,10,2,3,0,2</td>
<td>72+5=77, 101+10=111, 108+2=111, 108+3=111, 111+0=111, 33+2=35</td>
<td>Monoo#</td>
</tr>
<tr>
<td>Monoo#</td>
<td>-5,-10,-2,-3,0,-2</td>
<td>77-5=72, 111-10=101, 110-2=108, 111-3=108, 111+0=111, 35-2=33</td>
<td>Hello!</td>
</tr>
</tbody>
</table>

Now that we know how one time pads work we can start implementing them!

### 2.1.2 Using Classes and Inheritance

This is your first experience creating your own classes! Get excited! They are a powerful idea that you will use throughout the rest of your programming career. If you need a refresher on how classes and inheritance work, refer to the lecture and recitation notes.

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

- **Message** contains methods that both plaintext and encrypted 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 generating a one time pad or encrypting a message.
- **EncryptedMessage** contains methods that are specific to a ciphertext, such as a method to decrypt a message given a one time pad.

### 2.2 Message

We have provided skeleton code in the Message class for the following functions. Your task is to fill in the methods of the Message class found in ps4b.py according to the specifications in the docstrings. Please see the docstring comment of each function for more information about the function’s specification.

- `__init__(self, input_text)`
- `get_text(self)`
- `shift_char(self, char, shift)` This should return a string containing `char` shifted by `shift` according to the method described above. Some hints to keep in mind:
  - `ord(char)` returns the ASCII value of a string `char` which contains a single character
  - `chr(ascii_num)` returns a string with the single character specified by `ascii_num`
  - We are only interested in the 95 ASCII characters from ASCII values 32 to 126
  - The `modulo operator` `%` which returns the remainder of division is helpful for "wrapping around".
- `apply_pad(self, pad)` This should return a string containing the ciphertext of `self.message_text` after pad is applied
We have also implemented a `__repr__` function so that when you print out your `Message` objects, the console displays a nice human readable result. **Please do not change it.**

2.2.1) Testing

You can test out your code so far by running `test_ps4bc_student.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_encrypted_message`. We will implement code for those in the next section.

2.3) PlaintextMessage

Again, your task is to fill in the methods of the `PlaintextMessage` class found in `ps4b.py` according to the specifications in the docstrings.

- `__init__(self, input_text, pad=None)`
  - You should use the parent class constructor (using `super()`) in this method to make your code more concise. Take a look at Style Guide #7 if you are confused.
  - The syntax `pad=None` indicates an optional argument that can be omitted and the specified default value of None is passed in instead.
    - For example, `PlaintextMessage('test')` and `PlaintextMessage('test', [0,15,3,9])` are both valid constructors but the former should generate a random pad and the latter should use the specified pad.
  - You should save a copy of `pad` as an attribute and not `pad` directly to protect it from being mutated.
- `generate_pad(self)`
  - Note: our `shift_char` should work for any arbitrary integer but for simplicity we'll only generate pads with integers in the range `[0, 110]`
  - Hint: random.randint(a, b) returns a random integer N such that a ≤ N ≤ b.
- `get_pad(self)`
  - This should return a copy of `self.pad` to prevent someone from mutating the original list.
- `get_ciphertext(self)`
- `change_pad(self, new_pad)`
  - Make sure `self.get_ciphertext` uses the new pad!

We have also implemented a `__repr__` function so that when you print out your `PlaintextMessage` objects it returns a nice human readable result. **Please do not change it.**

2.3.1) Testing

You can test your new class by running `test_ps4bc.py`. You should now be able to pass all the tests starting with `test_message` and `test_plaintext`.

2.4) EncryptedMessage

Given an encrypted message, if you know the one time pad used to encode the message, decoding it is trivial. That's because if we shifted a character by x to encrypt it then to decrypt it we simply shift it back by −x! Therefore to decrypt a message that was shifted with the onetime pad [i, j, k, ...] we simply apply the pad [−i, −j, −k, ...]. So if $\textit{sip!}$ is the encrypted message, and [5, 1, 7, 2] was the pad used to encrypt the message, then [−5, −1, −7, −2] decodes the encrypted message and gives you the original plaintext message.

<table>
<thead>
<tr>
<th>Ciphertext</th>
<th>One Time Pad</th>
<th>Process</th>
<th>Plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\textit{sip!}$</td>
<td>5,1,7,2</td>
<td>36−5=126, 105−1=104, 112−7=105, 33−2=126</td>
<td><del>hi</del></td>
</tr>
</tbody>
</table>

We now will implement decryption in the `EncryptedMessage` class.

Fill in the following methods of the `EncryptedMessage` class found in `ps4b.py` according to the specifications in the docstrings.

- `__init__(input_text)`
  - As with `PlaintextMessage`, use the parent class constructor `super()` to make your code more concise. Take a look at Style Guide #7 if you are confused.
  - `decrypt_message(self, pad)`
    - This function should be very short. Using the above explanation of how to decrypt a one-time pad, try to use a function you already wrote in the `Message` class.

2.4.1) Testing
3) Part C: Using Your Classes

Now that we have created our classes in ps4b.py we will learn how to use them in ps4c.py! At the top of ps4c.py you can see the line:

```python
import ps4b # Importing your work from Part B
```

This imports your code from the ps4b.py file if it is in the same folder. To use a class created in Part B you can initialize it like this: my_message = ps4b.Message("My Message!").

There are a few helper functions we have implemented for you: load_words, is_word, and get_story_string. These will be helpful for implementing decrypt_message_try_pads and decode_story. You don't need to understand exactly how they work, but you should read their associated docstrings to understand what they do and how to use them.

3.1) Decoding Ciphertexts

One time pads are secure so we can't find a plaintext message without the pad used to encrypt it. However, if we have a ciphertext and a list of the pads which might have been used to encrypt it, we can find which pad actually encrypted it and what the plaintext message is.

To do this programmatically we would try decrypting the ciphertext with each pad in the list and count the number of English words in the output. We can count the number of words by splitting the decrypted output on spaces and testing if each is a valid English word. Then we can assume that pad that produces a plaintext message with the most valid words was the pad used to encrypt the message. Additionally, in the event of ties, we want to return the last pad that results in the maximum number of valid English words.

**Fill in decrypt_message_try_pads(self, pads) using the approach described above.**

You may find the helper functions is_word(wordlist, word) and load_words, and the string method split useful. *Note that is_word will ignore punctuation and other special characters when considering whether or not a word is valid.*

3.1.1) Testing

You should now be able to pass all the tests in test_ps4bc_student.py.

Please note that in the student tester your decrypt_message_try_pads will rely on your work from ps4b.py. However, the staff tester will have one test where it will use our reference staff implementation of ps4b.py. Therefore **you should make sure that you are using your getters and setters rather than directly accessing class attributes from decrypt_message_try_pads**!

3.1.2) Decoding a Story

Bob is trying to share a story with Alice without us being able to learn the story. Fortunately for us we overheard Bob when he was sharing all of his one time pads with Alice; we just don't know which pad he used for his story.

**Implement decode_story to find what Bob’s story was.**

- use get_story_string to get the ciphertext of Bob's story and get_story_pads to get a list of Bob's one time pads.
- Use your decrypt_message_try_pads function to find what Bob’s message to Alice was.

Check the output of decode_story by uncommenting the code at the bottom of ps4c.py and running ps4c.py. This function is **not tested** in test_ps4bc_student.py, but be prepared to discuss the decrypted story at your checkoff.

4) Hand-in Procedure

4.1) Time and Collaboration Info

At the start of each file, in a comment, write down the names of your collaborators. For example:

```python
# Problem Set 4B
# Name: Jane Lee
# Collaborators: John Doe
```
4.2) Half-way Submission

All students should submit their progress by the half-way due date (1 week before the final due date).

This submission will be worth 1 point out of the problem set grade and will not be graded for correctness. The intention is to make sure that you are making steady progress on the problem set as opposed to working on it in the final days before the due date.

You may upload new versions of `ps4a.py` until Nov 09 at 09:00PM. You cannot use extensions or late days on this submission.

Please refresh the page before submitting a new file. If you do not, your latest submission won't be updated.

4.3) Final Submission

Be sure to run the student testers `test_ps4a_student.py`, and `test_ps4bc_student.py` and make sure all the tests pass. However, the student tester contains only a subset of the tests that will be run to determine the problem set grade. Passing all of the provided test cases does not guarantee full credit on the pset.

You may upload new versions of each file until Nov 16 at 09:00PM, 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.

When you upload a new file with the same name, your old one will be overwritten.

4.3.1) Part A

4.3.2) Part B

4.3.3) Part C