Assignment 2: A Treemap Visualiser

Learning goals

After completing this assignment, you will be able to:

- model different kinds of real-world hierarchical data using trees
- implement recursive operations on trees (both non-mutating and mutating)
- implement an algorithm to generate a geometric tree visualisation called **treemap**
- use the `os` library to write a program that interacts with your computer's file system
- use inheritance to design classes according to a common interface

Introduction

As we've discussed in class, trees are a fundamental data structure used to model all sorts of hierarchical data. Often, a tree represents a hierarchical categorization of a base set of data, where the leaves represent the data values themselves, and internal nodes represent groupings of this data. Files on a computer can be viewed this way: regular files (e.g., PDF documents, video files, Python source code) are grouped into *folders*, which are then grouped into larger folders, etc.

Sometimes the data in a tree has some useful notion of size. For example, in a tree representing the departments of a company, the size of a node could be the dollar amount of all sales for that department, or the number of employees in that department. Or in a tree representing a computer's file system, the size of a node could be the size of the file.

A **treemap** is a visualisation technique to show a tree's structure according to the weights (or sizes) of its data values. It uses rectangles to show subtrees, scaled to reflect the proportional sizes of each piece of data. Treemaps are used in many contexts. Among the more popular uses are news headlines, various kinds of financial information, and computer disk usage. Some free programs use treemaps to visualise the size of files on your computer; for example:

- (Windows) WinDirStat, (OSX) Disk Inventory X, (Linux) KDirStat

For this assignment, you will write an interactive treemap visualisation tool that you can use to visualise hierarchical data. It will have a general API (implemented with inheritance, naturally!) and you will define specific subclasses that will allow you to visualise two different kinds of data: the files and folders in your computer, and a categorization of Computer Science Education research papers. We will make the following simplifications for this assignment:

1. The order in which subtrees are shown **matters when modelling files, but not when modelling research paper data**.

2. The colours in the visualization are randomly-generated independently for each node.

Setup and starter code

The zip file you received contains all the code you need for A2.

Unzipping this file should give you the following files and folders: - `a2_sample_test.py` - `cs1_papers.csv` - `example-directory` (a sample folder for you to test on if needed) - `papers.py` - `print_dirs.py` - `tm_trees.py` - `treemap_visualiser.py`

Problem description

The program for this assignment consists of two main components:

1. A data modeller that takes in some hierarchical data source (such as, but not limited to, a folder in your computer) and represents it in a tree data structure. This component will primarily be written by you.
2. An interactive graphical window showing a *treemap visualisation* of this data to the user. This component is primarily provided to you, and contains client code that uses your data modeller.

Please read through the following sections carefully to understand what is required for this program. Then we'll break it down into tasks for you to do.

Data modeller: abstract version

In order to use a treemap to visualise data, we first need the actual data to be stored in a tree. For this program, we have provided an `TMTree` class to define a common interface. Our treemap visualiser will expect for any sort of data that we want to visualise to have that same interface. To visualise any hierarchical dataset, we simply need to define a subclass of `TMTree`.

The `TMTree` class is similar to the basic `Tree` class that you have already worked with, with a few differences. The three most conceptually important ones are discussed here, but there are others you'll explore when you dig into the code.

First, the initializer is considered private, and should be *overridden* by each subclass. This is because these trees will be initialized through custom procedures that depend on the data we want to represent. (So a tree representing files on your computer will be created in a different way than a tree representing research paper data. You'll do both on this assignment!)

Second, the class has a `data_size` instance attribute that is used by the treemap algorithm to determine the size of the rectangle that represents this tree. `data_size` is defined recursively as follows:

- If the tree is empty (i.e., its `_name` is None), then its `data_size` is 0.

- If the tree is a single leaf, its `data_size` is some measure of the size of the data being modelled; e.g., it could be the size of a file, or the number of citations for a paper.
- If the tree has one or more subtrees, its `data_size` is equal to the sum of the `data_size`s of its subtrees.

Third, we'll also track some information inside our tree related to how to display it. For now, we'll just say that there is an attribute which is called `_expanded`; we'll discuss it more when we talk about the dynamic visualization of the tree.

This is a little abstract. Let's now discuss one concrete type of hierarchical data that you will be responsible for modelling on this assignment.

Data modeller: file system example

Consider a folder on your computer (e.g., the `assignments` folder you have in your `csc148` folder). It contains a mixture of other folders ("subfolders") and some files; these subfolders themselves contain other folders, and even more files. This is a natural hierarchical structure, and can be represented by a tree as follows:

- Each *internal node* corresponds to a *folder*.
- Each *leaf* corresponds to a *regular file* (e.g., PDF document, movie file, or Python source code file).

The `_name` attribute will always store the **name** of the folder or file (e.g. `preps` or `prep8.py`), not its path, e.g. `/Users/courses/csc148/preps/prep8/prep8.py`.

The `_data_size` attribute of a leaf is simply how much space (in bytes) the file takes up on the computer. The `_data_size` of an internal node - remember, representing a folder - corresponds to the size (in bytes) of all files contained in the folder, including its subfolders. (Computer folder sizes are actually calculated a little differently than adding up the sizes of the files and subfolders contained in them, but we'll use this simplified idea of folder size for this assignment.)

In code, you will represent this filesystem data using the `FileSystemTree` class, which is a non-abstract subclass of `TMTree`.

Visualiser: a dynamic visualization

Your program will allow the user to interact with the visualiser to change the way the data is displayed to them. As you'll see later in this handout, the user will be able to expand or collapse nodes in the tree, just as you can double-click on folders in your computer's file browser to expand them and see the files inside, or navigate back up a level to see those files collapsed back into a single folder. At any one time, parts of the tree will be displayed and others not, depending on what nodes are expanded.

We'll use the terminology **displayed-tree** to refer to the part of the data tree that is displayed in the visualiser. It is defined as follows:

- The root of the entire tree is in the displayed-tree.
- For each expanded tree in the displayed-tree, its children are in the displayed-tree.

The leaves of the displayed-tree will be the trees whose rectangles are displayed in the visualization.

We will assume that (1) if a tree is expanded, its parent is expanded, and (2) if a tree is *not* expanded, then its children are *not* expanded.

Note: the displayed-tree is not a separate tree! It is just the part of the data tree that is displayed.

Visualiser: the treemap algorithm

The next component of our program is the treemap algorithm itself. It takes a tree and a 2-D window to fill with the visualization, and returns a list of rectangles to render, based on the tree structure and `data_size` attribute for each node.

For all rectangles, we'll use the **pygame representation of a rectangle**, which is a tuple of four integers `(x, y, width, height)`, where (x, y) represents the upper-left corner of the rectangle. Note that in pygame, the origin is in the upper-left corner and the y-axis points *down*. So, the lower-right corner of a rectangle has coordinates (x + width, y + height). Each unit on both axes is a pixel on the screen.

There are many variations of the treemap algorithm. For this assignment, we'll use a dynamic version – one that lets us see different visualizations of the same tree. However, the process for generating the tree is always the same, even as the displayed-tree changes.

For simplicity, we'll use "size" to refer to the `data_size` attribute in the algorithm below.

The `update_rectangles` method should work like so:

Input: A data tree that is an instance of some subclass of `TMTree`, and a pygame rectangle (the display area to fill).

Output: None - It will directly modify the `.rect` attributes of the tree objects inside the data tree.

Algorithm:

Note: Unless explicitly written as "displayed-tree", all occurrences of the word "tree" below refer to a data tree.

1. If the tree has size 0, then it's rect should have an area of 0.
2. If the tree is a leaf in the displayed-tree, then it should take up the entire area of the given rectangle.

3. Otherwise, if the display area's width is greater than its height: divide the display area into vertical rectangles, one smaller rectangle per subtree of the displayed-tree, in proportion to the sizes of the subtrees.

   Example: suppose the input rectangle is (0, 0, 200, 100), and the displayed-tree for the input tree has three subtrees, with sizes 10, 25, and 15.

   ```
   - The first subtree has 20% of the total size,
     so its smaller rectangle has width 40 (20% of 200): (0, 0, 40, 100).
   - The second subtree should have width 100 (50% of 200),
     and starts immediately after the first one: (40, 0, 100, 100).
   - The third subtree has width 60, and takes up the remaining space:
     (140, 0, 60, 100).
   ```

   Note that the three rectangles' edges "touch" each other, but do not overlap, this means that any single point/coordinate on a displayed-tree can only be occupied by a single rectangle.

   You can assume that every non-zero-sized subtree will have a large enough size that its computed rectangle has a width and height >= 1.

4. If the height is greater than or equal to the width, then make horizontal rectangles instead of vertical ones, and do the analogous operations as in step 3.

5. Recurse on each of the subtrees in the displayed-tree, passing in the corresponding smaller rectangle from step 3 or 4.

Note about rounding: because you're calculating proportions here, the exact values will often be floats instead of integers. For all but the last rectangle, always truncate the value (i.e., round **down to the nearest integer**). In other words, if you calculate that a rectangle should be (0, 0, 10.9, 100), round the width down to (0, 0, 10, 100). Then a rectangle below it would start at (0, 100), and a rectangle beside it would start at (10, 0).

However, the *final* (rightmost or bottommost) edge of the last smaller rectangle should *always* be equal to the outer edge of the input rectangle. This means that the last subtree might be a bit bigger than its true proportion of the total size, but it won't be a noticeable difference in our application.

You will implement this algorithm in the `update_rectangles` method in `TMTree`.

Visualiser: displaying the treemap

The code in treemap_visualiser.py runs the treemap algorithm, and then uses pygame directly to render a graphical display to the user. You will only edit this file to uncomment or make minor edits to the client code for the `TMTree` class.

The pygame window is divided into two parts:

- The treemap itself (a collection of colourful rectangles).
- A text display along the bottom of the window, showing information about the currently selected rectangle, or nothing, if no rectangle is selected.

Every rectangle corresponds to one subtree (with `data_size` not zero) in the displayed-tree. If the whole tree has `data_size` zero (so no rectangles are returned by treemap), the entire screen should appear black.

Visualiser: user events

Note: Unless explicitly written as "displayed-tree", all occurrences of the word "tree" below refer to a data tree.

In addition to displaying the treemap, the pygame graphical display responds to a few different **events**:

a. The user can close the window and quit the program by clicking the X icon (like any other window).

b. The user can **left-click on a rectangle** to select it. The text display updates to show two things:

   - the names on the path from the root of the data tree to the selected leaf. The names are separated by a *separator string* defined in the subclass.
   - the selected leaf's `data_size`

   Clicking on the same rectangle again unselects it. Clicking on a different rectangle selects that one instead.

The following two events allow the user to actually mutate the data tree, and see the changes reflected in the display. The original data is not changed - we are just using the visualization as a tool for *simulating* changes on a dataset, and seeing what would happen. Once the user performs one of these events, the tree is no longer in sync with the original data set.

All changes described below should **only change the tree object**; so if a rectangle is deleted or its size changes, **DO NOT** try to change the actual file on your computer!

c. If the user presses the Up Arrow or Down Arrow key when a rectangle is selected, the selected leaf's `data_size` increases or decreases by 1% of its current value, respectively. This affects the `data_size` of its ancestors as well, and the visualisation must update to show the new sizes.

   Details:

   - A leaf's `data_size` cannot decrease below 1. There is no upper limit on the value of `data_size`.

- The 1% amount is always *rounded up* before applying the change. For example, if a leaf's `data_size` value is 150, then 1% of this is 1.5, which is rounded up to 2. So its value could increase up to 152, or decrease down to 148.

d. If the user selects a rectangle that is a leaf in the whole tree, then hovers the cursor over another rectangle that is an internal node in the whole tree and presses `m`, the selected leaf should be moved to be a subtree of the internal node being hovered over. If the selected node is not a leaf in the whole tree, or if the hovered node is not an internal node in the whole tree, nothing happens.

e. If the user selects a rectangle that is either a leaf or internal node and then presses `Del`, as long as it has a parent (i.e. it is not the root node), then that node should be deleted by removing it from the parent's subtrees (and hence the visualization).

The next six events change the displayed-tree but *not* the underlying data tree.

f. If the user selects a rectangle, and then presses `e`, the tree corresponding to that rectangle is expanded in the displayed-tree. If the tree is a leaf, nothing happens.

g. If the user selects a rectangle, and then presses `c`, the parent of that tree is unexpanded (or "collapsed") in the displayed-tree. (Note that since rectangles correspond to leaves in the displayed-tree, it is the parent that needs to be unexpanded.) If the parent is None because this is the root of the whole tree, nothing happens.

h. If the user selects a rectangle, and then presses `a`, the tree corresponding to that rectangle, as well as all of its subtrees, are expanded in the displayed-tree. If the tree is a leaf, nothing happens.

i. If the user selects any rectangle, and then presses `x`, the entire displayed-tree is collapsed down to just a single tree node. If the displayed-tree is already a single node, nothing happens.

j. If the user selects any rectangle, and then presses `q`, the selected rectangle will replace the current displayed-tree.

k. If the user presses `b` at any time, the selected rectangle's parent tree (if any) will replace the current displayed-tree.

Task 0: Getting started

0. Review your worksheet and notes from the release activity in lecture.
1. Watch the following video to see some of the operations displayed above: A2 demo video
2. Download the starter code into your folder for Assignment 2.

3. Open `tm_trees.py`, and read through the starter code carefully. There's quite a bit of information contained in the docstrings.
4. Open `treemap_visualiser.py`. There's less code in here, but more of it will look unfamiliar because of the pygame library.
5. Run `treemap_visualiser.py`. You should see a black window appear on your screen, which you can also close to exit the program. There will initially be no functionality, but as you progress through the assignment, this visualiser will begin to display some cool things.

Task 1: Representing the file system

In the starter code, we have provided both an incomplete version of `TMTree`, and a skeleton of the `FileSystemTree` class that could be used to subclass it and model how much space your files take up on your computer.

You should do three things here:

1. Complete the initializer of `TMTree` according to its docstring. This will get you familiar with the attributes in this class.
2. Complete the initializer of `FileSystemTree`. This method should be implemented recursively – in other words, part of this method should involve calling the `FileSystemTree` initializer to build subtrees.
3. Read the abstract methods from `TMTree` and the overridden counterparts from `FileSystemTree`.

To complete step 2, you will need to read the Python `os` module documentation to learn how to get data about your computer's files in a Python program. We recommend using the following functions:

- `os.path.isdir`
- `os.listdir`
- `os.path.join`
- `os.path.getsize`
- `os.path.basename`

The file `print_dirs.py` is a small sample program that works with the `os` module. Feel free to derive your own implementation from this.

You may **not** use the `os.path.walk` function - it does a lot of the recursive work for you, which defeats the purpose of this part!

Warning: don't use string concatenation (`+`) to try to join paths together. Because different operating systems use different separators for file paths, you run the risk of using a separator that is invalid on the DH lab machines on which we'll test your code. So instead, make sure to use `os.path.join` to build larger paths from smaller ones.

You should add the subtrees in your `FileSystemTree` in the order they are produced from `os.listdir`.

Progress check!

After this step is done, you should be able to instantiate a `FileSystemTree` by calling the initializer on a path, and then inspect the tree's contents in the debugger. **Write doctests and pytests** to test the attributes of your `FileSystemTree` objects; having these tests now will save you a lot of time and energy in later tasks.

Task 2: Implementing the treemap algorithm

Your next task is to implement the `update_rectangles` method in `TMTree`, which is the actual method responsible for computing the rectangles for the treemap representation of a tree. This method should update the rectangle stored in each `TMTree` that represents that tree - it is responsible for making sure every tree has a rectangle that reflects its data size. Your implementation must match the treemap algorithm described above.

Once again, your approach should be recursive; make sure you understand the role of the `rect` parameter before writing any code.

We *strongly* recommend implementing the base case before starting the recursive step, to give yourself a check that you understand what the method is supposed to do.

For the recursive step, a good stepping stone is to think about when the tree has height 2, and all leaves have the same `data_size` values. In this case, the original rectangle should be partitioned into equal-sized rectangles.

Warning: make sure to follow the exact rounding procedure in the algorithm description above. If you deviate from this, you might get numerical errors what will lead to incorrect rectangle bounds.

Once you have completed `update_rectangles`, you should complete the method `get_rectangles` to produce a list of all of the rectangles in the tree. Your approach here should be recursive as well. This method should not modify the tree, only return the rectangles to display it. Together, `update_rectangles` and `get_rectangles` implement the treemap algorithm.

The docstring of this method refers to the displayed-tree. For now, you can assume the whole tree is the displayed-tree - in a later task you will make the change to implement the displayed-tree.

Progress check!

The basic treemap algorithm does not require pygame at all: you can check your work simply by instantiating a `FileSystemTree` object, calling `update_rectangles` on it, then calling `get_rectangles` and checking the list of rectangles returned. This is one way we'll be testing your code. **Write your own doctests and pytests** for `update_rectangles` and `get_rectangles`.

This is the best way to check edge cases (e.g., making sure you're rounding properly).

In the main block of `treemap_visualiser.py`, put in a path from your own computer (or use the default one if you wish). After that, take a look at the method `render_display` in the same file and follow the `TODO` comments by uncommenting some code. Try running the program: if all goes well, you should see a *beautiful* rendering of your chosen folder's files in the pygame window!

Task 3: Selecting a tree

The first step to making our treemap interactive is to allow the user to select a node in the tree.

To do this, complete the body of the `get_tree_at_position` method. This method takes a position on the display and returns the leaf corresponding to that position in the treemap.

In Task 5, you will return to this method and modify it to select a leaf in the displayed-tree, but for now, write the method so that it returns a leaf in the *whole* tree.

Progress check!

Try running `treemap_visualiser.py` again, and then click on a rectangle on the display. You should see the path to that file displayed at the bottom of the screen. Click again to unselect that rectangle, and try selecting another one.

Task 4: Making the display interactive

Now that we can select nodes, we can move on to making the treemap more interactive.

Inside the `event_loop` function in the `treemap_visualizer` module, we have provided a skeleton of a loop that repeatedly waits for a new event, and then processes the event based on its type.

To support these events, write the bodies for the `TMTree` methods `change_size`, `update_data_sizes`, `move`, and `delete_self`.

Note that the `change_size`, `move`, and `delete_self` methods should implement the behaviours described earlier in this handout. The `update_data_sizes` method is called in multiple scenarios in the pygame code.

Progress check!

One of the nice things about code with an interactive display is that it's usually pretty straightforward to test basic correctness. Try running the treemap visualiser, and see if you can resize, move, and delete rectangles. Make sure the text display always updates when the selected rectangle changes.

Write doctests and pytests for the methods you created in `tm_trees.py`. Keep in mind that these are certainly possible to test, since they shouldn't involve pygame at all.

Task 5: Implementing the displayed-tree

Now you will implement the displayed-tree functionality for the visualizer, so that the user can expand and collapse their view of the data. Start by editing the initializer for `TMTree` to set the initial value of `_expanded` to `False` for all trees.

Then, make the necessary changes to `get_rectangles` and `get_tree_at_position` so that they produce their results based on the displayed-tree, not the whole data tree.

Next, write the methods `expand`, `expand_all`, `collapse`, and `collapse_all` and make necessary changes to previous methods that will be required to implement the displayed-tree functionality.

Use the client code in `event_loop` to tell you what the public interface should be for these methods. It is up to you to design, document, and implement these methods. You may not, however, add any new attributes (public or private) to the `TMTree` class, nor modify the interface of any of the existing methods. You are welcome to add private helper methods.

Also keep in mind that after completing this task, one of the tests in `a2_sample_test.py` will fail.

Progress check!

Try running the treemap visualiser. You should initially see a single rectangle. See if you can click on it and expand and collapse rectangles. Make sure the text display always updates when the selected rectangle changes.

Write doctests and pytests for the methods you created in `tm_trees.py`. Keep in mind that these are certainly possible to test, since they shouldn't involve pygame at all.

Task 6: One more data set

Once you have fully implemented your `TMTree` class, you'll take advantage of the fact that it is for any hierarchical data by writing another class that inherits from it, and that represents a dataset involving categorized Computer Science Education research papers.

If you have not yet completed Tasks 1-5, we recommend you don't even look at this task or its files yet (trust us - it will probably confuse you). You should not be making any assumptions about them in your `TMTree` code - in fact, we

could even test your `TMTree` by creating our own subclass of it, and testing the visualizer with that subclass.

The new dataset

This second dataset contains information about research papers on computer science education. (If you have an interest in education, you might enjoy taking a look at some of them. If you are on a UofT computer, you have free access to them all.) All of the papers fall under the broad category `CS1`, which is the term used in the literature for an introductory programming course for beginners, like our CSC108. The file is in csv format, and its contents are mostly self-explanatory. In the Category column you'll find a string describing of the category that the paper falls into, in the format

```
category:sub-category:sub-sub-category:...
```

From this, we can infer a tree of categories, and that's what you'll be visualizing. For example, suppose these were all the category strings in a data file:

```
a:b:c
a:b:d
m:n
m:o:p
a:b:d:e
a:f
```

Categories `a` and `m` are the top-level categories given, and they both fall under the overarching category `CS1`. Draw the tree of categories for the data above.

There is one extra feature we've added to class `PaperTree`: the initializer allows client code the option of making publication year be the first layer of subtrees below the `CS1` root before dividing further by categories and subcategories. A parameter to the initializer controls this. Note that if we do organize by year, portions of the category and subcategory tree will be repeated under each year node. (Which portions depends on which categories and subcategories appear in the dataset for that year.)

Your task

Take a look at these two files for task 6:

- `papers.py`
- `cs1_papers.csv`

NOTE: if you want to look at `cs1_papers.csv`, we recommend you open it up in a tab in PyCharm, or be careful to not save any changes if you open it with Excel or another spreadsheet program.

Decide on, and document, new instance attributes that are necessary in order to implement `PaperTree`. Then complete the class by overriding the initializer

from `TMTree` – that's all you need to do! We have suggested two helper functions that you may find useful. These create a dictionary representing the dataset and then build the tree from it. In the dictionary, each key is a category and its value is another dictionary of the same kind, representing all of its descendant categories and their papers.

Progress check!

After you've completed the tasks in `papers.py`, you should be able to run `treemap_visualiser.py` with `run_treemap_papers()`.

And you guessed it: you can write tests for this as well!

Testing

As part of this assignment, we also expect that you write tests for the code you submit. These tests will not be graded, but they will be the basis for any remark requests. (i.e. you can choose not to submit them, but you will also not be able to submit a remark after you receive your grades). In order to submit a valid remark request, you must also have submitted tests that cover the scenario in which you are submitting a remark request for.

Your tests should be submitted in the file `a2_my_tests.py`.

Polish!

Take some time to polish up. This step will improve your mark, but it also feels so good. Here are some things you can do:

- Pay attention to any violations of the "PEP8" Python style guidelines that PyCharm points out. Fix them!
- In each module you will submit, run the provided PyTA call to check for flaws. Fix them!
- Check your docstrings to make sure they are precise and complete and that they follow the conventions of the Function Design Recipe and the Class Design Recipe.
- Look for places where you can make improvements to your code to make it more efficient. You shouldn't sacrifice readability, but you shouldn't have your program doing lots of unnecessary work either.
- Read through and polish your internal comments.
- Remove any code you added just for debugging, such as print statements.
- Remove any `pass` statement where you have added the necessary code.
- Remove the `TODO's` wherever you have completed the task.
- Take pride in your gorgeous code!

Submission instructions

1. **DOES YOUR CODE RUN**?! Does it pass your thorough test suite (the additional tests you have to write, not just the sample tests)?
2. Login to MarkUs and find A2.
3. Submit the files `tm_trees.py`, `papers.py` and `a2_my_tests.py`.
4. On any DH lab machine, make a new folder and download all of the files you submitted into it. Test your code thoroughly. *Your code will be tested on the DH lab machines, so it must run in that environment.* This step will also confirm that you **submitted the right version** of the required files, and didn't **introduce an error at the last moment**, for example by adding a comment of changing a variable name.
5. Congratulations, you are finished with your last assignment in CSC148! You are now one step closer to being a wizard/witch who masters parser-tongue. :)