Online Protein Splicer

By: Jordan Henry

Demo

Random Sequence: KHHSRSALMIATKYSHQFQDHVLLPNQMRPLQVSYMPTISAFEFCEFEGC

Pre-Planning

Hint: Object Oriented Programming.

- I figured that the step of actually generating the table of cutswould be the most work (and code).
- The enzyme's only job is to say where to cut. The actual cutting is the same.
- Given that a child class can inherit methods from its parent, and that these methods can definitely reference material which is different in the child classes:
    - I decided to create a module in which the parent class contains a method by which the table of cuts is generated.
    - The child classes should all inherit this method, and feed into it their own list of cut sites, based on their own logic.

Rough Outline: (Changed A Few Times)

1. Make a generic class that is a “splice”, which will contain as datamembers the raw sequence and all the user inputs (but not enzyme type), and will generate the splice table as a method. -- It should come with a generic list of splice sites, just so everything will run.
    a. Generate the starting table, cutting at all possible splice sites, ignoring the missed cleavages condition. Why: Because the entries generated by missing cleavages are essentially combinations of the original table of splices. [I thought recombining them would be simple…}
    b. Generate the fully fleshed out table results, based on the missed cleavages condition, and recombining the original entries.
    c. Generate one last table for display, removing entries from the second table by length % weight conditions.

[I thought that the “three” tables design would be most straightforward-- if not efficient--way to solve problem.]

2. Create child classes.
3. Package in TurboGears for input/output.

Code Commentary: Splice_Mod.py

1. **Parent Class. Generates table for output.**
   Note: Every argument except ’rawseq’ comes with default values, making them optional. (Later became redundant.)

2. **Arbitrary list of splice sites.**

3. **Last position in sequence is added to splice list.** This is now a list of all “milestones”.

4. **Create a fresh ‘entry’ each time. Fill it.** Added to it in order 4.1. Start-stop position. 4.2 Molecular weight. 4.3 Missed cleavages, which will equal zero for all these entries in the first table. 4.4 Left of sequence 4.5 The sequence itself. 4.6. Right of sequence. -- Get to use a try-except block to catch where the right-of-seq doesn’t exist, because the end of the splice is… the end.

5. A negative index as ‘i’ would cause it to work from the back of the splice list. It also would not produce an error message. Glad I caught that.

6. **Entry becomes a new row in the table**.

Code Commentary: Splice_Mod.py

7. **Second table is a bit convoluted.** It takes entries in the first table, and fills in entries in between them in the response to the ‘missed cleavages’ condition.

8. **Screen out any blank entries added to the table**, in the process of bringing it over, with condition mass = 0.

9. **Now to build the missed cleavages table.** General strategy is combining an entry with those before it, based on the ‘mc’ condition. Since there is nothing before the first entry, just add it.

10. **Starting from the second entry:** For each entry, add the original to the new table.

11. **‘Add_entry’** was meant to be a placeholder. It’s the mc=1 condition for each entry. It **combines values from two entries.** It takes the starts and end of both splices and merged them. 11.2 It adds the weights together. 11.3 It uses mc = 1, for obvious reasons. 11.4 - 11.6. I takes the left of seq from the first, and right of seq from the second, and adds together their actual sequences.11.7 Inefficient, but I simply **checked the mc condition. If it’s not zero, then we add this entry.**

12. **Add matrix.** The real work is done by creating a matrix of entries which iterate on each other. In this matrix, the previous mc = 1 condition (add_entry) simply becomes the first entry. 12.1 Appends a new (blank) row to the add_matrix table, and fills it by a.adding the material from the previous row, and b. Modifying it using the information from a row in the splice_table which is ‘j’ rows back. Note finally the ‘if i > (j+1):’ condition. This is to prevent the creation of an “impossible” row. Eg. Taking the 2nd peptide, and combining it with the previous two. -- **After all these values are updated, the new row of the add_matrix table is also added to the ‘MC_table’.**

13. **‘LW_table’ simply takes everything in the ‘MC_table’, and cuts out any entries which are disqualified by the length and weight conditions.**

14. **Child classes are all quite straightforward.** In their __init__ method, they define the list of splice sites according to their own logic. They all start with ‘self.splice_list = [0]’, and append based on the sequence. They then inherit the table generating method from the parent class.

Splice_Mod.py {What I would do differently....}

- General Idea: All the information you need is contained in the a. raw sequence b. List of user supplied parameters. c. Start and end points of each splice.

  So:

1. Take the list of splice sites, inclusive of ends: (eg. [0, 10, 20]), and use the missed cleavages condition to create a list of pairs, containing all possible start and end points. (eg. [[0,10],[0,20],[10,20]]).
2. Use each pair in the list (of pairs) to generate an entry.
3. Based on length and weight constraints, decide whether to add it to the table.

Note: Could filter by length in an earlier step, but I prefer to keep both filtering steps in one place.

Code Commentary: Splice_App.py

Splice_App.py: Purpose is simply to receive all possible arguments for Splice_Mod.py, with one additional argument (what enzyme should be used), call the appropriate child class, feed it the arguments and receive the table result.

root.py:

1. **A search form, for input.** It takes all the same inputs as the ‘dice’ template, so that the search action can be ‘/dice’. So, it goes directly to that template.
2. **Actually generates information.** For display on the ‘dice’ template. It does so by utilizing ‘Splice_App’, which itself utilizes ‘Splice_Mod’.

dice.xhtml:

Simply creates a table, using iteration, and results provided.

What I learned:

The best thing is an extremely detailed outline. The sooner you can start figuring out the most efficient way to solve the problem, the less likely you are to go with a more labour intensive solution than necessary, in order to get through it.