S1 Challenge

Sobre o Projeto

Bem, acredito que só pela pasta já deu pra estranhar esse projeto, mas tenho uma explicação, rs. Vou começar explicando a escolha de linguagens.

Eu nunca usei RoR até o dia 26/01, porém tenho uma certa base em Python, entretanto, base insuficiente para fazer um layout bacana para submissão de dados e exibição de resultados. Como recomendaram usar RoR como framework bacana para esse tipo de coisa, decidi tentar aprender pelo menos o básico nesse meio tempo. De fato, consegui aprender bastante coisa em dois dias, mas só o que consegui foi rodar um server com algumas páginas, algumas funcionalidades (novo livro, editar livro e botões de voltar) e um pouco de layout, além de algumas sutilezas que entrarei em detalhes na Parte II.

Não quero dar desculpa pelo projeto incompleto, mas pensei em pedir mais tempo não só porque eu nunca trabalhei com RoR, mas porque tive problemas técnicos em casa depois que um raio caiu no bendito roteador da internet e me deixou na mão durante dois dias. De qualquer forma, resolvi fazer mesmo assim e entregar o que está ao meu alcance por hora!

Pensei no desafio como em etapas, na primeira o serviço recebe os dados do cliente (livros + regras de ordenação), na segunda (e mais difícil), que resolvi fazer em Python porque conheço melhor a sintaxe, é onde ocorre o processamento desses dados: separar id, título, autor e edição e comparar e ordenar cada um desses atributos de acordo com a direção desejada para cada um. Depois disso, teríamos a etapa 3, o output; nessa etapa eu gostaria de poder ter conseguido integrar meu código Python em RoR (pesquisei sobre inclusive, porém precisaria de mais tempo para implementar) para poder exibir os resultados num server, porém esse resultado acabou sobrando para o terminal do PyCharm mesmo (ou no terminal onde tiver rodando o código .py).

Parte I - Python

A parte em Python, como disse, consiste na segunda etapa, o processamento de dados fornecidos pelo cliente. Esses dados idealmente seriam recebidos pelo layout em RoR, mas para fazer meu código decidi receber os dados a partir de um arquivo externo, cliente.txt. Nesse arquivo temos uma lista de livros com seus respectivos títulos, autor e edição e temos a regra de ordenação, podendo variar atributo (título, autor, edição) e direção (ascendente ou descente), sendo ordenação nula simplesmente o espaço vazio.

Ok, agora vamos ao código! A primeira linha é de praxe para evitar problemas de caracteres especiais, como o acento circunflexo.

Ao invés de fazer o código gerando e lendo arquivo, decidi usar diversas funções por uma questão de desempenho e legibilidade. A primeira função processa o arquivo **cliente.txt**, lendo, salvando numa variável de dados e separando entre livros e ordenação. Logo em

seguida temos duas funções similares: uma organiza quais são as regras de ordenação, que chamei de ‘ordem’, e a outra processa a parte de livros para podermos organiza-los na próxima função. A fim de organizar os livros, identifiquei padrões no texto, como ‘-‘ e vírgulas para separar os atributos. Então, criei uma lista vazia para poder adicionar nela dicionários dos livros, sendo a primeira informação o index, a segunda o título, a terceira o autor e a quarta a edição; retorno uma lista organizada de livros com suas devidas informações separadas.

Agora vem a parte mais chatinha do código que é a ordenação. Separei as ordenações de acordo com quantos atributos são levados em conta: zero ou um (**ordenar_zeroum()**), dois (**ordena_2()**) e três (**ordena_3()**). Ao final, reuni as três funções em uma só para que haja somente um chamado. ou um atributo das outras, por ser mais fácil e ficar mais claro.

A fim de facilitar a localização do index dos atributos, criei um dicionário de index, depois é só o definir dentro de cada função de acordo com a ordem submetida. Na primeira função basta identificar se temos um ou nenhum atributo, em caso de nenhum, retorno a própria lista inserida. Já em caso de haver um atributo, identifico se é “ascendente”, caso seja, ordeno a lista de acordo com o atributo; caso não seja, o inverso ocorre e apenas utilizo a função reverse.

As próximas duas funções seguem o mesmo raciocínio. O que elas fazem é comparar títulos e autor (funcionando por eliminação quanto a edição) e retornar um valor. Caso os títulos não sejam afetados pela ordenação, então o que vale para ordena-los é o autor, e caso o autor também empate, somente aí a edição entra como critério. Cria-se então uma listona L que usará a função **sorted()** com um parâmetro key para especificar a função a ser chamada em cada elemento da listona antes de fazer as comparações. Para interpretar as funções de comparações que utilizei, que retornam números negativos, positivos ou zero, utilizei a função **cmp_to_key()**, essa foi a forma que encontrei de transformar as funções de comparação em chaves e conseguir dar um retorno inteligível para o código.

Na hora de processar a lista final, na função **ordenar()**, é preciso

Ao final, crio uma função para reunir os diferentes tipos de ordenação, de forma a não precisar utilizar duas funções diferentes para mesma tarefa.

É importante que no arquivo **cliente.txt** todas as direções estejam em letra minúscula e espaçadas normalmente em vírgulas. Também vale notar que eu importei alguns módulos para me ajudar: **string, operator** e **functools**. O código provavelmente não ficou muito padronizado ou comum (deu uma trabalheira, na verdade) porque não achei uma função mais simples que ordena para múltiplos atributos, apesar de ter procurado, então acabou ficando mais “braçal” mesmo e, talvez por isso, mais complexo, porém fiz o possível para deixa-lo compreensível dentro do tempo proposto.

Parte II – Ruby on Rails

Essa parte foi um grande desafio para mim porque eu nunca havia trabalhado com RoR antes, de forma que devorei alguns tutoriais, vídeos e cursos, mas, é claro, não pude fazer magia. Bem, algo que me ajudou muito e eu gostei bastante foi a sintaxe, além de muito

similar ao Python, permite também incorporar outras linguagens além de Ruby, como HTML. Além disso, as facilitações em criar um server, controllers e modelos são muito interessantes, não precisamos ficar criando muitas pastas e arquivos manualmente.

Gostei também das operações **CRUD** e dos princípios de **Don't Repeat Yourself (DRY)** e **Convention Over Configuration (COC)**. Pude ver um pouco de como DRY se aplica quando os códigos de criar um novo livro e edita-lo eram praticamente o mesmo, de modo que compensaria criar um outro arquivo ***_form.html.erb*** contendo a parte repetida e depois apenas embuti-lo nos arquivos de novo livro (***new.html.erb***) e edição de livro (***edit.html.erb***), deixando o código mais limpo e simples.

Uma dificuldade que encontrei, da qual acredito ser razoável de resolver, foi criar um campo de texto ou número para a Edição do livro:

```
<% @books.each do |book| %>
  <tr>
    <td><%= book.title %></td>
    <td><%= book.text %></td>
    <td><%= book.text %></td>
    <td><%= link_to 'Edit', edit_book_path(book) %></t
  </tr>
<% end %>
```

```
<p>
    <%= form.label :edition_year %><br>
    <%= form.text_area :number %>
</p>
```

Em preto tenho meu arquivo (1) ***views/books/index.html.erb*** e em vermelho tenho (2) ***views/books/_form.html.erb***. O problema foi usar a mesma referência de texto em (1) para Autor e Edição, não pegando a referência do número submetido em (2), a Edição, mas sim do seu texto acima, o Autor. Assim, autor e edição se confundem no layout.

O outro problema, e o que me levou a optar pelo Python, foi conseguir processar o input pelos botões e caixas que criei abaixo da "Forma de Ordenação", essa seria a etapa 2, que só consigo resolver em Python no momento.

Essa segunda parte fica mais como um ensaio do que consegui aprender nesse pouquinho tempo com meu interesse pela área. O desafio de fato está na parte um, apesar de não cumprir certos requisitos e não ser apresentado numa interface bonita.