ECE453/CS447/ECE653/CS647/SE465
Design of Distribute Django app (v1)

Patrick Lam and Lin Tan

This document describes the design of the `distribute` app.

Background information

If one should happen to collect assets and is hoping to save for retirement, it would be wise to invest these assets. There are arguments that a "passive" approach, holding a mixture of index funds, is best for most of us [1][2]. In such an approach, you need to decide on an asset allocation. Plus, there are various types of accounts (in Canada: taxable, TFSA, RRSP) and some assets work best in certain accounts. Which assets should live in which accounts?

In any case, you can think of `distribute` as a simple constraint solver for a restricted domain. It distributes a pool of money among a number of holdings which live in various accounts.

1 Model

The `distribute` app consists of two main calculations: one which distributes assets to various holding types according to a specified allocation, and one which allocates holding types across accounts.

1.1 Distribution

The first part of the `distribute` calculation decides how much of which assets to hold. The relevant parts of the model are the *holding types* and the *holding type proportion rules*. Here's an example of a (sensible) set of holdings and rules:

Holding types. While there are many index funds, two sets of important features are whether they hold "equities" or "bonds", and which country they are indexing. I have created four types:

- equity/CA
- equity/US
- equity/INTL
- bond/CA

[1] `http://web.stanford.edu/~wfsharpe/art/talks/indexed_investing.htm`
[2] `http://www.marketwatch.com/story/warren-buffett-to-heirs-put-my-estate-in-index-funds-2014-03-13`

Holding type proportion rules. The main choice to make is how much to hold in each of these funds. This associates a type with a percentage. An example proportion is:

- equity/CA: 20%
- equity/US: 20%
- equity/INTL: 20%
- bond/CA: 40%

Calculation 1: Optimal raw allocations. Given a set of proportion rules and a total number of dollars, the `calculate_raw_allocations` function calculates an optimal dollar amount for each holding. The total number of dollars is the sum of the dollars in the holdings plus the user-provided input to the HTML form. The easiest way to investigate the behaviour of the holding type proportion rules is in a database with no holdings (which is also the initially-provided database). Enter a total amount in the input field, and press "calculate". The app will show the optimal allocation.

1.2 Allocations (to accounts)

The other important set of constraints comes from the fact that you can hold money in different accounts, and some holdings are better to hold in some accounts than others (due to tax law). Furthermore, the government limits amounts that you can add to each account. In this part of the app, I track the amounts in each account and compute an optimal allocation of holdings across accounts.

Accounts. The app supports three kinds of accounts: RRSPs, TFSAs, and non-registered accounts. I have three accounts, one of each kind. RRSPs and TFSAs have contribution limits. (Keeping track of them is beyond the scope of this app; we will assume that the amount of money in these accounts is fixed at the initial holdings.) Reflecting that constraint, I've unchecked the "can add money" box for RRSPs and TFSAs in the provided database. Each account also has a name.

Holdings. To reallocate existing holdings, the app needs to store the existing holdings. A holding consists of an amount (currency units are uniform; say Canadian dollars), an account, and a holding type. (Ignore the purchase date).

For each account with "can add money" set to false, the app calculates account limits based on holdings already in the account. Accounts with "can add money" set to true can accept additional holdings.

Holding location rules. As alluded to above, certain holdings do best in certain accounts. (For instance, bonds are subject to unfavourable tax treatment in non-registered accounts). The holding location rules encode my understanding of the optimal location for each holding. The app greedily interprets the rules, one at a time, to arrive at an allocation of holdings to accounts.

Rules are numbered with a priority (re-using priorities produces undefined behaviour) and then identify a holding type and an account where that holding type should be held, until the account limit.

Calculation 2: Allocation algorithm. Allocation, implemented in `apply_allocation_rules`, is simple:

for all lr ∈ location rules **do**
  for all a ∈ accounts matching lr.account_type **do**
    allocate max(amount-remaining-to-invest-in-holding-type, room-remaining-in-a)
      to a new holding of lr.holding_type in *a*
  end for
end for

The implementation tracks the remaining amounts with Python dicts (also known as associative arrays) and updates them as it allocates amounts.

1.3 Miscellaneous helper functions

The two calculations above are the crux of the app; I recommend testing them. There are a couple of helper functions:

- `calculate_maxima`: calculates the maximum amount allowed in an account
- `calculate_current_holdings`: calculates the holdings currently in an account (for display purposes)
- `apply_allocations_to_accounts`: inserts data from the `allocated_amounts` dict (as calculated by `apply_allocation_rules`) into the appropriate holdings in the `current_holdings` dict (for display).

2 Test requirements

We are looking for a set of test requirements that will exercise the system's behaviour. The only input that the view accepts is a number of dollars to add (or subtract) from the allocation. However, to meaningfully test the system, you must also generate various model entities for the two algorithms I've described above.

3 User's manual

The virtual machine that you've created for this assignment includes the `distribute` app in the `q4` subdirectory. It comes with a minimal database already (installed via a fixture). The username for this database is `root` and the password is `password`. Run and test the system the same way that you ran and tested the `isin` sample application. You don't need to specify `--settings=isin.test_settings` for test runs.

You can add to that database through the web admin interface, at `http://localhost:4567/admin` from your host computer (i.e. your laptop); or you can visit the front-end at `http://localhost:4567/track`.

(When you actually create test cases, you can either create new fixtures or insert data programmatically. Hint: use `python manage.py dumpdata` to create a fixture from database contents.)