Exercise: Detailed Distributional Analysis

Summary

This exercise examines the detailed distributional impacts of the tax policy in the earlier microsimulation calculation.

Input Data

File **households.csv** is the earlier file giving data on 1000 households. As before, it has 5 columns: `id`, `type`, `inc`, `a`, and `b`. The `type` variable wasn't used before but it will be important in this exercise. It ranges from 1 to 4 and indicates the demographic type of the household. Here we'll think of it as indicating the region of the country where the household resides. In other studies it could be used to indicate other characteristics, such as the race, gender, or age of the household's head, the size of the household, whether the household lives in an urban or rural area, and so on. In those cases there would be many more than four types.

File **quantities.csv** gives the quantities demanded by each household under the base case and tax policy simulations. It has three columns, one for the household's ID and one each for the household's demands under the base and policy cases: `id`, `qd1` and `qd2`. In case you're curious, it was produced by running the earlier `ind_demand()` function for each equilibrium price and then writing out the output. However, you do not need to do that for this exercise: you should use the provided **quantities.csv** without recalculating it.

Deliverables

Please prepare a script called **dist.py** that calculates the effective tax rate (ETR) for each household and then reports the median ETR for the groups indicated below. Then update **results.md** replacing the TBD placeholders with your answers to the questions in the file.

Instructions

To help make the structure of the analysis clearer, the instructions below have been divided into sections labeled A, B, etc. However, that's just for clarity: you should build a single script that does all of the steps.

A. Initial setup

1. Import `csv`.

2. Import `numpy` as `np`. Numpy is a large library of functions for numerical analysis and is traditionally imported as `np` to allow function calls to be brief.

3. Include the line below to import function `defaultdict()` from the `collections` module.

   ```
   from collections import defaultdict
   ```

   As discussed in class, the `defaultdict()` function creates enhanced dictionaries that have a default value for new keys. It's handy because it eliminates the need for checking whether a key is already in a dictionary before trying to use it. The `from` statement is a special version of the `import` statement that allows its argument (here `defaultdict`) to be called without its module prefix. That is, the script below will call it as `defaultdict()` rather than as `collections.defaultdict()`.

B. Defining a function for reading the data

1. Define a function called `read_data` that takes two arguments: `filename` and `floatlist`. The `floatlist` argument will be used to indicate which fields in each record should be converted to numbers using `float()`. The function will read a CSV file and return a dictionary of the data. The body of the function should do the following:

   1. Open `filename` for reading using file handle `fh`.

   2. Set `reader` to the result of calling `csv.DictReader()`on `fh`.

3. Create an empty dictionary called `file_data`.
4. Use `hh` to loop over `reader`. Within the loop do the following:
    1. Use `field` to loop over `floatlist`. Within this inner loop, set `hh[field]` to the value of `float()` called on itself: `float(hh[field])`. That will be the only line in the loop.
    2. After the `field` loop finishes, set `this_key` to the value of the household's ID, which is in `hh['id']`.
    3. Set the value in `file_data` for `this_key` to `hh`.
5. After the `hh` loop finishes close `fh` and return `file_data`.

C. Reading the data files

1. After the function definition, begin the main code by setting `hh_data` to the result of calling `read_data()` on arguments `"households.csv"` and `['a','b','inc']`. That is, the call should have `"households.csv"` as the first argument and the list as the second argument.
2. Set `qty_data` to the result of calling `read_data()` on `"quantities.csv"` and `['qd1','qd2']`.

D. Joining the datasets

1. Now join the quantity data onto the household data using the ID of the household as the join key. To do that, use `hh_rec` to loop over the values of `hh_data`. Within the loop do the following:
    1. Set `this_id` to the ID for household in the current household record, which will be `hh_rec['id']`.
    2. Set `qty_rec` to the entry in `qty_data` for household `this_id`.
    3. Set `hh_rec['qd1']` and `hh_rec['qd2']` to the corresponding values from `qty_rec`.

    After the `hh_rec` loop completes, `hh_data` will contain all the information about each household from the two files.

E. Computing income quintiles and the ETRs

1. Now we'll extract the full list of incomes and compute where the breaks between income quintiles fall. To do that, create a list called `incomes` that contains the value of `hh_rec['inc']` for every household in `hh_data`. The easiest way to do it is to use a list comprehension but you can also do it by creating an empty list and then looping through the values of `hh_data` and appending each household's value of 'inc' to the list.
2. Next, set `inc_cuts` to the result of calling numpy's `percentile()` function on `incomes` and the list of percentiles that define the lower bounds of quintiles: `[0,20,40,60,80]`. After the call, `inc_cuts` will include 5 values: the minimum income for each quintile. That is, the first element will be the minimum income in the dataset, the second element will be the income at the 20th percentile, and so on. Be sure to note that there's no "s" at the end of the function name: it's `percentile()`, not `percentiles()`. The call should look like this:

    ```
    inc_cuts = np.percentile(incomes,[0,20,40,60,80])
    ```

3. Now we'll calculate the ETR for each household, and also record each household's income quintile while we're at it. Start by creating a variable called `pd1` that is equal to 53.35 and one called `pd2` equal to `55.27`. Then create a variable called `dp` that is equal to `pd2 - pd1`.
4. Next, use `hh_rec` to loop over the values of `hh_data`. Within the loop do the following:
    1. Set `hh_rec['rev']` to the revenue burden on the household, which is `dp*hh_rec['qd2']`.
    2. Set `hh_rec['etr']` to the household's ETR: 100 times `hh_rec['rev']` divided by `hh_rec['inc']`. Note that the value will be a percentage, so 1 represents 1%.

3. Set variable `quint` to 0. It will be used with the loop below to determine the income quintile for the household.

4. Use `min_inc` and to loop over `inc_cuts`. Within the loop include an if statement that checks whether `hh_rec['inc']` is greater than or equal (>=) to `min_inc`. If so, add 1 to `quint`. That's all that will be in the for loop.

5. After the `min_inc` loop, `quint` will be the quintile where the household's income falls (where the first quintile is 1). Set `hh_rec['quint']` to the string version of `quint`, which is obtained by calling `str()` on `quint`. Converting `quint` to a string is needed to facilitate printing the grouped results that will be computed below. See the tips section for details.

F. Grouping the results by type and quintile

1. After the `hh_rec` loop completes the next step is to aggregate the ETR results into three sets of groups: (1) by type and quintile, (2) by quintile over all types, and (3) by type over all quintiles. The first step is to create a variable called `grouped` to hold the grouped information. It will be a `defaultdict` that automatically creates an empty list the first time any given key is used. Use the call:

   ```
   grouped = defaultdict(list)
   ```

2. Now use `hh_rec` to loop over the values of `hh_data`. Within the loop do the following:

   1. Set `key_by_type_quint` to a tuple consisting of `hh_rec['type']` and `hh_rec['quint']`. This will be used to group the data by type and quintile, with one group for each combination of `type` and `quint`.

   2. Set `key_by_quint` to a tuple consisting of the string `"all"` and `hh_rec['quint']`. This will be used to group the data purely by quintile without distinguishing by type.

   3. Set `key_by_type` to a tuple consisting of `hh_rec['type']` and the string `"all"`. It will be used to group the data purely by type without distinguishing by quintile.

   4. Now use three statements to append `hh_rec['etr']` to the values of `grouped` for each of the three keys above. The statements will be very similar, differing only by the key. As you'll see below, this will let the script compute and report a wealth of data: the median ETR by type and quintile, the median ETR by type alone, and the median ETR by income quintile alone.

G. Computing the median ETR for each group

1. Now we'll compute the median ETR for each group of households. Start by creating an empty dictionary called `medians`. Then use variable `this_key` to loop through the keys of `grouped`. Within the loop, use `np.median()` call to calculate the group's median ETR from the list of individual ETRs in `grouped` as follows:

   ```
   this_median = np.median( grouped[this_key] )
   ```

   Then set the value of `medians` for key `this_key` to the value of `this_median` rounded to 2 decimal places.

H. Printing the results and writing them up

1. Print a simple header for the results along the lines of `"type quint etr"`.

2. Use `group_key` to iterate over the result of calling `sorted()` on the keys of `medians`. Within the loop use a print statement to write out `group_key[0]`, `group_key[1]` and `medians[group_key]`. The value of `group_key[0]` is the type and the value of `group_key[1]` is the income quintile.

3. Finally, look over the results and then use a text editor to fill in answers to the questions in **results.md**. Replace the "TBD" with your response. It's OK to be very concise: the goal is insights rather than a detailed exposition.

Submitting

Once you're happy with everything and have committed all of the changes to your local repository, please push the changes to GitHub. At that point, you're done: you have submitted your answer.

Tips

- This exercise goes through one of the fundamental workflows of data analytics: joining datasets, calculating detailed results from the joined data, grouping the results according to various attributes, applying one or more functions to the grouped data, and then reporting the results.
- The reason the quintiles needed to be converted to strings is to avoid errors when sorting the keys of `grouped`. The `sorted()` function can sort a list of numbers or a list of strings but it can't sort a mixed list. Since there will be values of `"all"` for some quintiles, the other quintiles have to be strings as well.