ChatGPT

Demographic Simulation Script – Detailed Breakdown

Re-Importing Libraries and Argument Parsing

At the top of the script, all necessary libraries are imported to ensure the script has access to required functionality (even if they were imported elsewhere, they are imported again for completeness). Common libraries likely include **pandas** (for Excel/JSON handling), **json** (for config files), **argparse** (for parsing command-line arguments), and possibly others like **numpy**. Re-importing in the script's main section guarantees that each run starts with a clean slate of library imports.

Following the imports, the script sets up an **argument parser** using `argparse`. This allows the script to be run from the command line with various options. For example, it probably defines arguments such as:

- `--config` or `-c` for the path to a configuration file (e.g. a JSON config).
- `--output` or `-o` for the output Excel filename.
- Perhaps arguments for scenario selection (like `--scenario` type or specific parameters).
- Other flags like verbosity or help text.

Using `argparse.ArgumentParser`, the script adds these arguments and then calls `parser.parse_args()` to retrieve the values. This way, a user can customize the simulation by providing different input files or scenario parameters when running the script. The argument parsing section also typically includes descriptive help messages so that running `-h` explains how to use the tool. In summary, this section prepares the environment and inputs for the simulation by loading libraries and reading user-provided arguments.

Progress Indicator: The `print_progress` Function

The script defines a helper function `print_progress()` to give visual feedback during the simulation's execution. This function likely takes parameters like the current step number and total steps (and possibly a task description) and prints an update to the console. For example, it might output: "Processing step 3 of 5...".

Internally, `print_progress` could calculate the percentage of completion and print a one-line progress message. In some implementations, it might overwrite the same line (using carriage returns) to update progress without flooding the console. In simpler form, it may just print a new line for each step. The purpose of this function is to inform the user that the simulation is running and how far along it is, especially since policy simulations can be time-consuming. Throughout the script, `print_progress` is called at key points (for instance, after loading data, after applying shocks, after calculations) to indicate which stage has been completed. This improves usability by making the script's workflow transparent in real time.

Configuration Handling: The `load_config` Function

The script uses a `load_config` function to manage configuration files. This function is responsible for reading a config file (commonly in JSON format, though it could be YAML or another format) and loading simulation parameters from it. Calling `load_config(config_path)` will typically:

- Open the config file from the given path.
- Parse its contents (e.g. using `json.load` into a Python dictionary).
- Validate or set default values for any missing entries.

The configuration file likely contains parameters such as file paths for input data, assumptions for scenarios (like percentage shock values or reallocation specifics), demographic constants, or any other tunable settings for the simulation. By using a config file, the script becomes flexible: users can change scenario parameters or input filenames without modifying the code.

Inside `load_config`, there may be error handling to ensure the file exists and is well-formed. For example, it might catch JSON parsing errors or missing file exceptions and inform the user. Once loaded, the config dictionary is returned and used to drive the rest of the simulation. In summary, `load_config` centralizes the handling of external configuration, making the simulation tool easier to configure and reuse for different scenarios.

The Core `run_simulation` Function

The heart of the script is the `run_simulation` function. This function orchestrates the entire simulation process, using the inputs and parameters to produce outputs. It can be broken down into several key stages:

Loading Baseline Data from Excel

The simulation begins by **loading baseline data** from an Excel file (or files). Using libraries like pandas, the script reads in data such as population figures, employment numbers, health indicators, etc., that serve as the starting point (baseline scenario). For example, it might use `pandas.read_excel()` to load a sheet with base year demographic data and maybe another sheet with parameter values.

If multiple sheets or files are involved, each is loaded and stored in a DataFrame or similar structure. Common data loaded at this stage could include: - **Baseline population by category** (age, sex, region, etc.). - **Baseline values of key variables** (e.g., employment count, incidence of a health condition, costs). - **Parameter tables** (like age distribution percentages, cost per case, prevalence rates, etc., if not provided via config).

This separation of data allows the simulation to reference real-world or assumed baseline conditions. The code identifies the relevant Excel file path from either the config or an argument and reads it. If the Excel file has specific sheet names for data and parameters, those are accessed accordingly. After this step, the program holds the baseline dataset and any supplementary parameters in memory for processing.

Shock Scenarios: Percentage Change, Reallocation, and Using 2012 Values

One core feature of this tool is the ability to apply different **shock scenarios** to the baseline. The script likely supports multiple scenario types, as indicated: - **Percentage Change Scenario:** This scenario applies a uniform percentage change to certain baseline values. For instance, if the scenario is a "10% increase in unemployment," the script would take baseline unemployment figures and increase them by 10%. The code might look conceptually like:

```
if scenario_type == "pct_change":
    for var in target_variables:
        new_value = baseline[var] * (1 + shock_percent)
```

Here, `shock_percent` could be provided in the config or as an argument (e.g., 0.10 for +10%). The script multiplies baseline values by (1 + percentage) to simulate growth or reduction.

- **Reallocation Scenario:** A reallocation shock redistributes values between categories. For example, a policy scenario might reallocate budget or population from one sector to another. In code terms:

  ```
  elif scenario_type == "reallocation":
      # Example: move X units from Category A to Category B
      baseline["CategoryA"] -= X
      baseline["CategoryB"] += X
  ```

  The script would reduce a certain variable by some amount and increase another, keeping totals constant or altering distribution. The details (which categories, how much) would come from the config or predefined logic. This scenario is useful for modeling policies where resources or people shift from one group to another.

- **Using 2012 Values Scenario:** This scenario uses actual values from the year 2012 as the shock input. In practice, that means the script will substitute certain baseline figures with those from 2012. Possibly the Excel data or config includes "2012" reference values. The code might do:

  ```
  elif scenario_type == "2012":
      for var in target_variables:
          new_value = reference_2012_values[var]
  ```

  Essentially, instead of modifying baseline by a percentage, it resets those variables to historical 2012 levels. This can simulate "what if conditions were like 2012 again" to compare against current baseline. It requires that the script has access to 2012 data – perhaps a separate sheet or a hard-coded dictionary loaded via config.

These scenario branches ensure the tool can flexibly simulate different types of shocks. The script will have logic to identify which scenario is requested (likely via a config field or CLI argument) and then execute the appropriate block to adjust the data.

Baseline Variables Identification

Before or while applying shocks, the script identifies **baseline variables** that are subject to change or of interest for output. Baseline variables refer to the key metrics in the dataset that represent the initial state (without any shock). For example, baseline employment count, baseline number of sick individuals, baseline costs, etc.

The code may maintain a list of these baseline metrics to easily reference them for calculations and for output comparisons. By isolating baseline variables: - The script can preserve their original values (for later comparison with scenario outcomes). - It can ensure only intended fields are modified by the shock scenario.

For instance, if the Excel data has many columns, not all will be altered by a given scenario. The script might filter out which columns correspond to baseline scenario values and which are calculated fields or unaffected constants. Those baseline fields then get copied or tagged so that after applying the shock, the script can calculate differences or just output both baseline and new values side by side. This identification step is important for clarity in results: users can see what the baseline was versus the new scenario outcome for each variable.

Applying Demographic Ratios (Parameter Data)

Many policy simulations incorporate demographic structure. Thus, the script applies **demographic ratios** from the parameter data to refine the calculations. This typically involves using known proportions or rates to break down aggregate numbers or to compute specific group metrics. Two common uses for demographic parameters are: - **Distributing Totals into Subgroups:** If the baseline data provides a total population or total count of something, the script might use age/sex ratio data to split that total into categories. For example, if baseline total population is 1,000 and parameter data says 60% are aged 18-64 and 40% are 65+, the script can calculate population_18_64 = 600 and population_65_plus = 400. This ensures the simulation's outputs are granular and demographically realistic. - **Applying Rates to Populations:** If the parameter data contains rates (like disease prevalence or labor participation rates) for different demographic groups, the script uses these to calculate absolute numbers. For instance, given a prevalence rate of 5% for a health condition in a certain age group and knowing the population of that group, it can compute the expected number of cases (population * 5%). Similarly, an employment rate applied to population yields number employed.

The script likely merges or aligns the baseline data with these ratios. For example, it might join a DataFrame of demographic breakdown percentages with the baseline totals and multiply accordingly. This step ensures that any simulation output reflects the underlying demographic patterns rather than treating the population as homogeneous. It's especially crucial after a shock scenario: if a shock increases the total population or a total count, the demographic breakdown can redistribute that increase appropriately across age and sex categories.

Modifying Variables According to the Shock Scenario

Once baseline values are loaded and prepared (and possibly broken into demographic detail), the script **applies the shock** to produce the new scenario values. This is the core transformation where the baseline scenario diverges according to the scenario assumptions:

- For **percentage change** scenarios, the script goes through each relevant baseline variable and scales it. Suppose one target variable is unemployment_count. If the shock is +10%, the script will set `unemployment_count_new = unemployment_count_baseline * 1.10`. This is done for all variables specified to undergo a percentage change. The reasoning is that a percentage shock mimics uniform growth or decline (e.g., a 10% budget cut or a 5% population increase).

- For **reallocation** scenarios, the script adjusts values between related variables. A concrete example: if the scenario is "shift 5% of people from unemployed to employed," the code will decrease the unemployed count by that amount and increase the employed count by the same amount (taking care not to go below zero). This maintains consistency (people aren't created or destroyed, just moved). The script likely calculates the absolute number to reallocate using the percentage and the baseline value, then updates the two (or more) categories accordingly.

- For **2012 values** scenarios, the script replaces current values with those from 2012. For instance, if baseline 2025 employment rate is 70% and 2012's was 65%, under this scenario it would use 65% as the employment rate for calculations. This might involve retrieving a stored 2012 dataset or hard-coded constants from config. The script ensures that any downstream calculation uses these 2012 values instead of the original baseline ones for the scenario run.

During this modification stage, the script carefully applies changes only to the intended variables, leaving others unchanged. It may create new variables or columns to store the post-shock values (for clarity, e.g., `employment_after_shock` vs `employment_baseline`). This separation allows later comparison of "before and after". The key reasoning here is to implement the scenario exactly – whether it's an across-the-board increase, a targeted shift, or a historical reset – so that the outcomes reflect that scenario's conditions.

Calculating Outcome Measures (Employment, Health, Costs)

After applying the scenario modifications, the script calculates various **outcome measures** to quantify the impact. These outcomes translate the raw changes into meaningful indicators: - **Employment-related outcomes:** If the simulation involves labor data, it might calculate total employed people, unemployment rate, labor force participation, etc. For example, given new employed and unemployed counts, the script can compute the unemployment rate = (unemployed / labor force) * 100. It might also compare it to the baseline unemployment rate. - **Health indicators:** In a health context, outcomes could include the number of cases of a disease, prevalence rate (cases per population), or healthcare coverage. For instance, if the scenario affects the number of people with a certain health condition, the script would calculate the new total cases and perhaps the percent of the population affected. - **Costs:** If cost data is part of the model (say healthcare costs or pension costs), the script will calculate total costs under the scenario. This typically involves multiplying quantities by unit costs from the parameters. For example, if there are X additional cases of an illness and the cost per case is Y (from parameter data), the total cost = X * Y. Similarly, for

employment, if fewer people are unemployed, perhaps benefit payout costs decrease – the script could calculate those savings by comparing baseline and scenario numbers.

The calculations are done using the updated scenario values (while possibly also computing the same for baseline to have a point of reference). The script might use vectorized DataFrame operations or simple arithmetic to derive these results. Each outcome measure is stored in a variable or added to an output dictionary/DataFrame. The reasoning is to turn the scenario's raw changes into policy-relevant metrics that stakeholders care about – percentages, totals, and rates that illustrate the scenario's effect.

Generating Detailed Metrics (Age-Sex Breakdown, Prevalence Rates)

Beyond overall outcomes, the script produces **detailed metrics** to give insight into how different groups are affected and how common certain outcomes are: - **Age-Sex Breakdowns:** Using the demographic ratios and group counts computed earlier, the script can break down outcomes by age group and sex. For example, it might produce a table of employment count by age group (18-24, 25-64, 65+ etc., for baseline vs scenario) and by gender. This answers questions like "which age group gains the most employment under this scenario?" or "how do health outcomes differ between males and females after the shock?".

To do this, the script likely iterates over each demographic category or uses group-by operations on the data. If earlier steps produced separate counts for each group, it might be as simple as formatting those into a report. If not, it may calculate them on the fly (e.g., applying age-specific rates to the scenario population of that age).

- **Prevalence Rates and Other Percentages:** Using the counts and the population numbers, the script calculates rates such as prevalence (cases per population) or participation rates. For instance, if it determined that 50,000 out of 1,000,000 people have a certain condition in the scenario, it will compute a prevalence of 5%. Likewise, it can compute the employment rate (employed divided by working-age population) or other ratios of interest. These rates provide normalized measures that are easier to compare than raw counts.

The generation of these detailed metrics is done for both the baseline and the scenario outcome, so that differences can be observed. The script might create new pandas DataFrames for these breakdown tables. Each row might represent a demographic group and columns for baseline count, scenario count, and perhaps the difference or percentage change. For rates, it may output baseline %, scenario %, and the change in percentage points.

Overall, this part of the script ensures the simulation results are not just a single number, but a rich set of data showing **who** is affected and **how**. This is vital in demographic/policy analysis, where impacts are often uneven across sub-populations.

Outputting Results to Excel and JSON

After all calculations are complete, the script prepares to **output the results** in user-friendly formats. It typically writes the results to: - **Excel File:** Excel is a convenient format for analysts, so the script likely creates an Excel workbook (using pandas `ExcelWriter` or similar) and writes multiple sheets. For example: - A sheet for **summary outcomes** (key metrics like total employment, total cost in baseline vs

scenario). - A sheet for **detailed breakdowns** (age-sex tables, etc.). - Possibly sheets for baseline data and scenario data for transparency.

Using pandas, this might look like:

```
with pd.ExcelWriter(output_path) as writer:
    summary_df.to_excel(writer, sheet_name="Summary", index=False)
    breakdown_df.to_excel(writer, sheet_name="Breakdown by AgeSex", index=False)
    # etc...
```

The script ensures that important indices are not dropped (or intentionally drops them for clarity) and that the data is neatly formatted (column headers, etc.). Writing to Excel allows end users to open the results in Excel and explore or visualize them.

- **JSON File:** In addition to Excel, the script writes results to a JSON file. JSON output is machine-readable and useful for integration with other tools or for programmatic access to the results. The script might dump a dictionary of results, for example:

  ```
  results = {
      "summary": summary_dict,
      "breakdown": breakdown_dict,
      # ...
  }
  json.dump(results, open(json_output_path, "w"), indent=2)
  ```

  Alternatively, it might use pandas' `to_json()` if the data is neatly structured. The JSON could contain similar information as the Excel (perhaps key metrics and breakdowns). This dual-output approach caters to different audiences: Excel for analysts and JSON for developers or further processing.

Throughout the output stage, the script handles potential errors (like if the file cannot be written) and ensures the file paths are taken from the arguments or config. By the end of `run_simulation`, the expected outputs (Excel and JSON files) are created on disk with all the calculated results.

Command-Line Interface via `argparse`

The script is designed to be run as a standalone tool, and the **command-line interface (CLI)** is powered by `argparse`. In the `if __name__ == "__main__":` block (or equivalent), the script uses the argument parser defined earlier to get user inputs. For example, a typical usage might be:

```
$ python simulate.py --config params.json --output results.xlsx --scenario
reallocation
```

When this command is executed: - `argparse` reads the `--config` argument, so `config_path` might be set to `params.json`. The script then calls `load_config(params.json)` to load all simulation settings. - The `--output` argument gives the base name for output files (e.g., `results.xlsx` and possibly the script might derive `results.json` from it). - `--scenario` specifies which scenario type to run (e.g. "reallocation"). The script would pass this into `run_simulation` or use it to select the shock logic.

Inside the code, after parsing, they likely populate variables or a dictionary for `run_simulation`. For instance:

```
args = parser.parse_args()
config = load_config(args.config)        # Load config file if provided
scenario = args.scenario or config.get("scenario_type")
output_excel = args.output or config.get("output_excel")
output_json = config.get("output_json")  # maybe defined in config or derived
```

Then the script would call:

```
run_simulation(config, scenario, output_excel, output_json)
```

passing in all necessary parameters to run the simulation. The CLI ensures that if a user does not provide a config file, required parameters might be supplied via other arguments (or defaults are used). It also makes the tool accessible without writing additional code – just run it with different flags for different needs.

The argparse interface often includes help text for each argument, making the tool self-documenting at the command line. This means a user can type `python simulate.py -h` and see descriptions of what each argument does (for example, *"--scenario: choose which shock scenario to apply (options: pct_change, reallocation, 2012)"*). This design shows the script is intended for flexible use, allowing different scenarios and configurations to be tested by passing arguments rather than editing the script.

Error Handling and Final Output

The script includes **error handling** to manage unexpected issues gracefully: - **Configuration and Input Errors:** If the config file path is invalid or the file is malformed, `load_config` will likely catch exceptions and print a clear error message (and possibly exit). Similarly, if the Excel data file is missing or cannot be read, the script will handle that (using a try-except around `pd.read_excel`). This prevents the script from simply crashing with a cryptic traceback; instead it provides the user a message like "Error: Could not open data file. Please check the path." - **Argument Validation:** The script may check that required arguments are provided. For instance, if no config is given, perhaps it requires explicit `--input-data` and other arguments. If something essential is missing, it can use `argparse`'s mechanisms to display an error or it can manually print a message and exit. - **Scenario Handling:** If an unknown scenario name is passed, the script might handle it by printing "Unrecognized scenario, must be one of [pct_change, reallocation, 2012]" and exiting, rather than proceeding with bad assumptions. - **Computation Errors:** During the simulation, there could be runtime errors (divide by zero, etc.). The code likely safeguards certain operations. For

example, if calculating a rate with a zero population, it might check for zero to avoid division errors, or at least handle the exception and set the rate to 0 or `None`. Memory errors or other unexpected issues might not be specifically handled, but the script is presumably written to work within normal data sizes.

At the end of the script's main execution (after `run_simulation` completes without errors), the **final output** stage is straightforward. The script will print a confirmation message to the console indicating success. For example: *"Simulation completed successfully. Results saved to results.xlsx and results.json."* This message confirms to the user that the process finished and where to find the outputs. It might also include timing information (how long it took) if the script tracks that, or any other pertinent note (like "Note: all results are in thousands of persons" if such scaling was used).

Finally, the program exits normally (returning control to the shell). The user is left with the generated Excel and JSON files containing a detailed breakdown of the scenario simulation. All the parts – from reading inputs, applying shocks, using demographic ratios, calculating outcomes, to writing outputs – work together to form a **demographic/policy simulation tool**. This tool allows analysts to plug in different scenarios (policy changes, demographic shocks, historical what-ifs) and quantitatively see the outcomes and impacts on various population segments and metrics. The clear structuring into functions (`load_config`, `run_simulation`, etc.) and the use of a command-line interface make it both maintainable for developers and usable for non-programmers who can run pre-defined scenarios by modifying config files or arguments. Overall, each component of the script contributes to a robust pipeline that takes in baseline data and assumptions, processes them step-by-step, and produces insightful results for decision-making.
