# "Shopping Cart" Project - Automated Testing Challenges

> Prerequisite: "Testing 1, 2, 3" Exercise

## Setup

From within your project's virtual environment (e.g. "shopping-env"), install the `pytest` package:

```
conda activate shopping-env # for example, if necessary
pip install pytest # (first time only)
```

After writing tests, you should be able to run them from the root directory of your project repository:

```
pytest
```

## Instructions

Read your existing project code. Think about ways to improve its readability and documentation. Think about ways to simplify and remove duplication. Think about decomposing the major responsibilities into component stand-alone functions (or maybe classes, if you prefer that kind of thing).

Think about the user experience and expectations. What do they need the program to do in order to consider it in "working condition"? What should the program do? Think of ways to express its desired functionality using common language.

Think of ways to verify your code is behaving as expected. Implement automated tests, for example using the `pytest` package. Feel free to reference the examples below, but it is not necessary to adhere to them exactly.

## Challenges

> NOTE: the testing prompts below are examples to help you think about what kind of functionality to test. Ultimately everyone's programs may operate differently. The overall goal is just to implement tests in your program, in whatever way best makes most sense for your individual circumstance. 😸

### Formatting Prices

Refactor price-formatting logic into a function called something like `to_usd()`, and implement a corresponding test called something like `test_to_usd()`.

Test various scenarios to ensure the price formatting function displays a dollar sign, two decimal places, and a thousands separator.

### Formatting Timestamps

Refactor timestamp-formatting logic into a function called something like `human_friendly_timestamp()`, and implement a corresponding test called something like `test_human_friendly_timestamp()`.

Test to ensure the function processes any given datetime object into a corresponding human-friendly string.

### Finding Products

Refactor product-finding logic into a function called something like `find_product()`, and implement a corresponding test called something like `test_find_product()`.

Test various scenarios to ensure the product lookup function finds and returns the proper product, even if the products are not sorted in order of their unique identifiers. What should happen when the function is passed a numeric identifier vs a string identifier? What should happen when there is no product matching the given identifier?

### Calculating Tax and Totals

Refactor subtotal and/or total price calculation logic into one or more function(s) called something like `calculate_total_price()`, and implement a corresponding test(s) called something like `test_calculate_total_price()`.

Test various scenarios to ensure the calculation function(s) produce the proper sum of prices, given a list of selected products and/or product identifiers.