NamedTuples and Immutability

Instructions:

1. Open up your `ProblemSetSkeleton` in vscode.
2. Create a new file called `immutability.py` and put it in the `skeleton` folder (`src/skeleton/immutability.py`).
3. Use the `immutability.py` file to run code and check your answers.

This worksheet is intended to illustrate how decision trees work and to provide more experience working with NamedTuples. Work through each task with the help of a partner (being sure to type/run each bit of code yourself).

Problem 0: Identify Yourselves

1. What are the names of the people in your group?

   >

2. Share this document with **everyone** in the group and make them editors.

Problem 1: Seems Easy

Use the code below to answer the following questions and complete the required tasks

```
def add(x: int) -> None:
    new_x = x + 1


if __name__ == "__main__":
    i = 5
    add(i)
    print(i)
```

BEFORE running the program, answer the following questions:

1. How do you think `x` gets its initial value when the `add()` function is called?

   >

2. What do you think happens to the `new_x` variable once the `add()` function completes?

   >

3. What do you think will be printed out when you run the program?

   >

NEXT, copy the code into your `immutability.py` file and run it.

1. Were your predictions correct? If not, what mistakes did you make?

   >

2. Whenever there's a problem this easy, a trick is sure to follow. What do you think the trick will be once we introduce data structures?

   >

3. Start fresh by deleting everything in your `immutability.py` file.

Problem 2: A Little Too Easy…

Use the code below to answer the following questions and complete the required tasks

```
def append(x: list[int]) -> None:
    x.append(3)
```

```
if __name__ == "__main__":
    l = [1, 2]
    append(l)
    print(l)
```

BEFORE running the program, answer the following questions:

1. How do you think `x` gets its initial value when the `append()` function is called?

   >

2. The function is declared to return `None`, yet it seems to modify `l`. Is this a contradiction? Why or why not?

   >

3. What do you think will be printed out when you run the program?

   >

NEXT, copy the code into your `immutability.py` file and run it.

1. Were your predictions correct? If not, what mistakes did you make?

   >

2. In what scenarios might the observed behavior be helpful?

   >

3. In what scenarios might the observed behavior be dangerous?

   >

4. Replace the `x.append(3)` line with `y = x.copy()` and `y.append(3)`. Run the program again and explain what the difference is.

   >

5. Start fresh by deleting everything in your `immutability.py` file.

Problem 3: Let me see you `id`

Use the code below to answer the following questions and complete the required tasks

```
def add(x: int) -> None:
    new_x = x + 1
    new_x_loc = id(new_x)
    print(f"The location of value stored in `add.new_x`: {new_x_loc}")


def append(x: list[int]) -> None:
    x.append(3)
    x_loc = id(x)
    print(f"The location of value stored in `append.x`: {x_loc}")


if __name__ == "__main__":
    i = 5
    i_loc = id(i)
    print(f"The location of the value stored in `main.i`: {i_loc}")
    add(i)
    print(i)

    l = [1, 2]
    l_loc = id(l)
    print(f"The location of value stored in `main.l`: {l_loc}")
    append(l)
    print(l)
```

```

**BEFORE** running the program, answer the following questions:

1. The `id()` function shows you **where in memory** a variable is stored. Why might this be useful?

   >

2. What do you expect to be printed for `id(new_x)` inside `add()` compared to `i` in the "main hook"?

   >

3. What do you expect to be printed for `id(x)` inside `append()` compared to `l` in the "main hook"?

   >

**NEXT,** copy the code into your `immutability.py` file and run it.

1. Were your predictions correct? If not, what mistakes did you make?

   >

2. Modify the `append()` function in the same way you did in the previous problem. What do you think `id()` will show you now?

   >

3. Copy your code for the **entire modified program** and paste it into the space below:

   ```

   ```

4. Start fresh by deleting everything in your `immutability.py` file.
   ```

Problem 4: NamedTuples

NamedTuples are really just fancy dictionaries. Your goal in this problem is to go from code that easily “mutable” to code that is harder to mess with.

```
def in_chicago(person: dict[str, str]) -> bool:
    if "chicago" in person["address"].lower():
        person["name"] = "Amy"  # Bug
        return True

    return False


if __name__ == "__main__":
    people = []
    people.append({"name": "Ted", "address": "123 Main Street, New
York"})
    people.append({"name": "Betty", "address": "18 Elm Street,
Chicago"})

    for person in people:
        lives_chicago = in_chicago(person)
        print(f"{person["name"]} is in Chicago: {lives_chicago}")
```

BEFORE running the program, answer the following questions:

1. The function contains a bug that modifies the person's name to `“Amy”`. Why might this kind of bug be particularly dangerous in a real system?

   >

2. What do you think will be printed out when you run the program?

   >

NEXT, copy the code into your `immutability.py` file and run it.

1. Were your predictions correct? If not, what mistakes did you make?

   >

2. If we had been using a `NamedTuple` rather than a dictionary, what would happen if we accidentally altered a person's name?

   >

3. Alter the code below so that it uses `NamedTuple` instead of dictionaries to represent people.

4. Copy your code for the **entire modified program** and paste it into the space below:

   ```

   ```

5. Start fresh by deleting everything in your `immutability.py` file.

Problem 5: Bad Programmer

An unsuspecting coworker is relying on you to create a simple function to help them look through a list of data to find the name of the oldest person. Fill in the `get_oldest()` function so that it produces the right output but also messes up their data so much they will never trust you again.

```
def get_oldest(l: list[list]) -> str:
    ...


if __name__ == "__main__":
```

```
    data = [["Corey", 17], ["Alex", 33], ["Gordon", 61], ["Sara", 2]]

    print(get_oldest(data))
    print(data)
```

BEFORE running the program, answer the following questions:

1. Prior to beginning to code, what are some creative ways you could 'mess up' their data while still returning the correct oldest person?

   >

2. What makes a list of lists particularly vulnerable to manipulation?

   >

NEXT, copy the code into your `immutability.py` file and start to alter it as described above.

1. Copy your code for the **entire modified program** and paste it into the space below:

   ```

   ```

Problem 6: Reflection

1. What is the difference between “primitives” and “data structures” when they are passed to functions?

   >

2. Primitives are “passed by value” and data structures are “passed by references. What do you think this means?

   >

3. Why is immutability a desirable property?

   >

4. Did you notice any obvious errors with this worksheet that should be fixed for future students? If so, what were they?

   >

5. Were there things in the worksheet that were overly confusing and that you think could be improved? If so, what were they?

   >