1. Social media channels

a) From tuples to classes

The first step is to replace the two tuples by classes. For example, you could do the following (using dataclasses):

```
@dataclass
class SocialChannel:
    type: str
    followers: int


@dataclass
class Post:
    message: str
    timestamp: int
```

The code now also needs to be updated in other places. For example, the `post_a_message` function needs to be updated to access channel properties instead of deconstructing tuples:

```
def post_a_message(channel: SocialChannel, message: str) -> None:
    if channel.type == "youtube":
        post_to_youtube(channel, message)
    elif channel.type == "facebook":
        post_to_facebook(channel, message)
    elif channel.type == "twitter":
        post_to_twitter(channel, message)
```

And in the `main` function, the list of channels now needs to be created using class initializer calls:

```
channels = [
    SocialChannel("youtube", 100),
    SocialChannel("facebook", 100),
    SocialChannel("twitter", 100),
]
```

Though doing this code change is not that hard, it does show one of the disadvantages of using low-level data structures like tuples: if you ever decide to use another representation for channels or posts, you need to change the code all over the place. This can of course also happen with classes, but then you can at least define the interaction mechanism yourself and build a sort of compatibility layer so that existing code still works without changing it too much. With tuples, this is a lot more challenging. You can find the full updated version of the code in `exercise_1_v2.py`.

b) Improving the post_a_message function

A nice way to solve the problem is by reconsidering whether a separate `post_a_message` function actually makes sense. In `exercise_1_v3.py`, I refactored the code so that it now uses an abstract class that defines a `post_message` method that then in turn call the lower level message posting function (in a real-life example, posting messages will probably be a lot more involved than simply calling a function though). Due to polymorphism, the correct message posting method is called automatically, so we no longer need the if-statement from the previous version.

A much shorter solution is given in `exercise_1_v4.py`. Here I'm not using abstraction at all and I moved to a mostly functional solution. I am defining a few simple types like `MessageSender` that serve as a kind of 'abstraction' in that they define the interface between the different areas of the program. The way I solved the need for the if-statement in this version is by putting references to the function in a dictionary, so you can simply map social channel type to the appropriate function in the `process_schedule` function.

What did your solution look like? Did you rely on classes or functions? How was it different from my solution?

2. Html Trees

a) Abstraction

What we need to do is create an `HTMLElement` abstract class. This class is basically what the `Div` class was before, but made abstract:

```
@dataclass
class HTMLElement(ABC):
    parent: HTMLElement | None = None
    x: int = 0
    y: int = 0

    def compute_screen_position(self) -> tuple[int, int]:
        if not self.parent:
            return (self.x, self.y)
        parent_x, parent_y = self.parent.compute_screen_position()
        return (parent_x + self.x, parent_y + self.y)
```

After that, each HTML element type needs to be a subclass of this class and you can then remove all of the duplication. For example, this is what the `Button` class will now look like:

```
class Button(HTMLElement):
    def click(self) -> None:
        print("Click!")
```

Take a look at what the new version of the code looks like in the attachment to this solution.

There's a couple of interesting things to note:

- The `Div` class is empty, so at the moment its only purpose is to mark that an HTML element is of type `Div`.
- `HTMLElement` is both a dataclass and an abstract base class. `Span` is a dataclass that inherits from `HTMLElement`. As you can see, dataclasses can properly deal with these kinds of inheritance relationships and generate the correct initializer methods.
- As you can see in the new version, you can now also compute screen positions of elements that are not divs, such as the span.

b) Protocols

The issue that you're going to run into is that with protocols you rely on structural typing (duck typing) instead of inheritance. That means that if you have an `HTMLElement` protocol and you define properties and methods inside of it, the other classes won't be able to use that if they don't inherit from `HTMLElement` anymore. There are a couple of ways to fix this:

- Even though protocols use structural typing, you can still inherit from them, just like with abstract base classes. You may think that's the best solution, but I personally prefer to keep structural typing 'pure' so that when you use protocols in your code, you know that you never have to inherit from them. If you still need to inherit from protocols in some cases, it may result in confusion: when do we need to inherit from them and when not?
- You could create two layers of inheritance. At the top layer is a protocol class that contains only the `compute_screen_position` method and properties for the parent, x, and y. You could then have an abstract base class (named `HTMLElementBase` for example), that follows the protocol's structure and then each specific HTML element type would then be a subclass of `HTMLElementBase`. In the areas of your application where you want to rely on structural typing, you can then use the `HTMLElement` protocol class, while still being able to provide a basic implementation of parents and positioning for the specific HTML elements.