1. Pure functions and side effects

a) `generate_id` is not a pure function because it relies on a random number generator. `weekday` relies on the current date and time. This means that the result of the function depends on factors outside of its own scope. Because of that, the function is harder to test.

In case of `generate_id`, one way to write tests for this is to not use unit tests, but property tests instead. This way, you can at least test some properties like making sure that the length of the generated id corresponds to what you passed along as a parameter. A second way to test this is to make sure the random number generator always has the same seed. You can then predict the ids that are being generated. It does require extra care though. You might break your tests simply by changing the order in which you run them!

In case of `weekday` you're going to have to patch the `today` function so that it always returns the same date and then you can test that the function returns the right weekday name. This is all doable, but it does require extra work to write tests due to these functions not being pure.

b) The file `exercise_1_solution.py` contains a possible solution that turns these functions into pure functions. As you can see, I used the same approach for both functions: instead of directly calling a function (like `random.choice`) or creating an object (like a `datetime` object), I used dependency injection to provide the function or object. Testing the code is now way easier, because in our testing code, we can supply the function or object we want.

Another change is that you see that the `main` function now serves as the place where everything is created and patched up. It's a really good idea to setup your application in such a way that there is one place where this happens - let's call that the "dirty" place. If your application looks like this, it means that all the parts have been properly separated and you can easily patch them together and change things independently.

Note: I used partial function application to make sure random.choice adheres to the `SelectionFn` type. Unfortunately, this is not correctly detected by the type system, so I wrote a comment behind the line to ignore type issues in this case.

2. Classes or functions

You can find both a object-oriented and function-based solution in `exercise_2_solution_oo.py` and `exercise_2_solution_fn.py` respectively. For the object-oriented version, I added a method to the class called `reset_to_factory` that does the job. In the functional version, I added a function.

The funny thing is that the difference between these two solutions is minimal. If you look at the function/method itself, they're exactly the same, except that in the object-oriented version the argument is called `self` and in the functional version it's called `laptop`. And in the main function, we either call the method with the dot syntax (in the object-oriented version), or we pass the object as an argument (in the functional version).

The syntax around classes and object is syntactical sugar. You can achieve the same thing with functions. In fact, even functions are syntactical sugar for lower-level coroutines. And if you go even lower than that, you'll end up at the basic Von Neumann computer architecture of the CPU that manipulates memory. That something is syntactical sugar doesn't mean that it isn't useful though. Sometimes classes are useful because you can structure information with them and group methods together with that information. Sometimes, functions are easier because they lead to shorter code and are often easier to test, especially if they're pure functions.