Computer Systems Modeling and Verification
(USEEN1)

Part II: A simple PBT framework

A simple PBT framework for Python

Property-based testing combines the design-by-contracts approach (where function specifications are based on properties) with random test data generation (since test cases do not require expected values). More specifically, given a function with a contract, we test the method by following these steps:

- a generator is used to create a series of random values for a given input type;
- the pre-condition is then used to filter the random input values for the function;
- the post-condition is finally used to check the ouput of each function invocation.

The generator, which needs to be composed with the pre-condition filter, is often called a *strategy* (or a *provider*) and the post-condition is usually just called the *property*.

Dedicated frameworks for property-based testing with Python, such as Hypothesis[1], provide a convenient way to express properties and to implement new strategies.

However, using Pytest parametrized tests[2] is sufficient in most cases: The builtin `pytest.mark.parametrize` decorator allows you to easily refer to a specific strategy. The series of random input values can be implemented as a `Sequence` (or, more efficiently, as an `Iterator`).

Consequently, in order to use Pytest as an effective PBT framework, we just need to implement some generators for the main Python datatypes and containers, and also to provide the tester with an API for developping his own generators.

The goal of this practical is to implement a such simple PBT framework based on Pytest.

Implementation

A generator for some type `T` can usually be implemented directly as a *generator expression* using a regular function (relying on the standard `random` module). Recall that the syntax of a generator expression is similar to list comprehension (except that we use parentheses instead of bracket and the result is an `Iterator` instead of a `Sequence`).

If this function is parameterless (also usually called a *supplier*), it can be used to build another generator. The type of a supplier can be defined as follows:

```
type Supplier[T] = Callable[[], T]
```

Module `strategies`, given below, provides generators for standard types and containers, defined as static methods (e.g. `arbitrary_int`, `arbitrary_str`, `arbitrary_list`, etc). Generators for custom types can then be created by combining these functions with constructors.

1. https://hypothesis.readthedocs.io/en/latest
2. https://docs.pytest.org/en/latest/how-to/parametrize.html

Generators expressions

For instance, this expression generates 16 arbitrary characters between `'A'` and `'F'`:

```
(arbitrary_char('A', 'F') for i in range(0, 15))
```

where `arbitrary_char` is one of the functions from module `strategies`.

Exercises

1. **Implementation**. Complete the implementation of module `strategy` by providing bodies to all functions (in place of the `TODO` comments). The specifications of these functions are given as informal comments in the module.

2. **Unit Testing.** Use the builtin `pytest.mark.parametrize` decorator from Pytest to test your generators.

3. **Examples.** Use your PBT framework to test a few functions. Recall that in order to test a function, you first need to define a property that characterizes the result of this function.