Computer Systems Modeling and Verification
(USEEN1)

Part II: Python libraries for testing

Several popular libraries should be considered when testing Python code. We will focus here on some of these libraries which are currently compatible the last version of Python:

- Pytest-cov library provides code coverage. An extension for VS Code, called Coverage Gutters, is also recommended in order to display the coverage within the IDE.
- Hypothesis is an advanced property-based testing library.
- Deal is a mature library for design-by-contract (which works well with Hypothesis).

These libraries (together with their dependencies) can be installed from the shell. First activate your conda environment (if required) and then run the following command from the folder containing the project files:

```
pip install -r requirements.txt
```

Using Pytest-cov

Pytest-cov is a plugin for pytest, it should be detected automatically. The plugin is in fact activated by the options given in the configuration file for the project (pyproject.toml). A new file called coverage.xml should be generated in the folder, and clicking on "watch" in VS Code satus bar should be enough to display the code coverage of the source files.

Using Hypothesis

Hypothesis relies on "strategies" in order to generate test data. These strategies can be chosen explicitely and even customized if required. In most cases however, the static type information (if provided), is enough to select a reasonable strategy.

For instance, to test the cat function, the annotation `@given(...)` tells Hypothesis to rely on the given type information, and thus to provide random strings for the test cases.

```python
@given(...)
def test_cat(left: str, right: str) -> None:
    result = cat(left, right)
    assert result.startswith(left)
    assert result.endswith(right)
    assert len(result) == len(left) + len(right)
```

Note. You can limit the number of test cases by adding the following annotation:

```
@settings(max_examples=10)
```

If you need to look at the generated values, you need to also add first this option:

```
@settings(max_examples=10, verbosity=Verbosity.verbose)
```

and then you should run command `pytest -s` directly from the terminal (option `-s` is required to prevent the capture of the output by `pytest`).

Using Deal

This section is adapted from Deal documentation about Contract-Driven Development.

Can we make it even simpler? Not really. The implementation can produce some values, and the machine can infer some properties of the result. However, someone else must say which properties are good and expected, and which are not. However, there is something else about our properties that we can do better. At this stage we have type annotations and, to be honest, they are just kind of properties. Annotations say "the result is a text", and our test properties clarify the length of the result, it's prefix and suffix. However, the difference is type annotations are the part of the function itself. It gives some benefits:

1. The machine can check statically, without the actual running of the code.
2. The human can see types (think "possible values set") for arguments and the result.

And Deal can make the same for function properties:

```
from deal import ensure

@ensure(lambda left, right, result: result.startswith(left))
@ensure(lambda left, right, result: result.endswith(right))
@ensure(lambda left, right, result: len(result) == len(left) + len(right))
def cat(left: str, right: str) -> str:
    return left + right
```

Ensure is the shining star of property-based testing. It works perfect for complex task when checking result correctness (even partial checking only for some cases) is much easier than the calculation itself.

You can also specify preconditions and postconditions.

Precondition: condition that must be true before the function is executed.

```
@pre(lambda xs: all(x > 0 for x in xs))
def sum_positive(xs: list[int]) -> int:
    return sum(args)

sum_positive([1, 2, 3, 4])
# 10

sum_positive([1, 2, -3, 4])
# PreContractError
```

Postcondition: condition that must be true after the function was executed.

```
@post(lambda result: result > 0)
def always_positive_sum(xs: list[int]) -> int:
    return sum(args)

always_positive_sum([2, -3, 4])
# 3

always_positive_sum([2, -3, -4])
# PostContractError
```

Post-condition allows you to make additional constraints about a function result. Use type annotations to limit types of results and post-conditions to limit possible values inside given types.

Remark. `@ensure` should be used for more general postconditions that accept not only the result, but also the function arguments.

Exceptions

`@raises`: specifies which exceptions the function can raise.

`@reason`: checks the condition if the exception was raised.

```
@raises(ZeroDivisionError)
@reason(ZeroDivisionError, lambda a, b: b == 0)
def divide(a: int, b: int) -> int:
    return a // b
```

Generating test cases

Deal can also generate random test cases using Hypothesis. Here is a complete example:

```
@raises(ZeroDivisionError)
@reason(ZeroDivisionError, lambda a, b: b == 0)
@pre(lambda a, b: a >= 0 and b >= 0)
@ensure(lambda a, b, result: a == result * b + a % b)
def divide(a: int, b: int) -> int:
    return a // b


@cases(divide, count=100)
def test_divide_contract(case):
    case()
```

Note. It is possible to display the result of the test cases as follows:

```
@cases(divide, count=100)
def test_divide_contract(case):
    result = case()
    print(result)
```

and then again, you should run command `pytest -s` directly from the terminal (displaying the full test case is also possible[1] but less trivial).

1. https://deal.readthedocs.io/basic/tests.html?highlight=index_of#practical-example