**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.
- Hypothesis is an advanced property-based testing library.

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:

```
pip3 install -r requirements.txt
```

## Using Pytest-cov

Pytest-cov is a plugin for pytest, it should be detected automatically. The plugin behaviour can be customized using the configuration file for the project (pyproject.toml). A new file called coverage.xml should be generated in the project folder, and VS Code should then be able to display the code coverage for the source files (note that installing an extension for VS Code, called Coverage Gutters, was required for previous versions of the IDE).

## 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 provide random strings for the test cases.

```
from hypothesis import strategies as st

@given(st.text(), st.text())
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)
```

**Remark.**

- By default, Unicode strings are used, but you can restrict them to `ascii` strings as follows:

  `st.text(st.characters(codec='ascii'))`

  and even to the Unicode category of "letters" as follows:

  `st.text(st.characters(codec='ascii', categories=(['L'])))`

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

  `@settings(max_examples=10)`

  If you also want to display the generated values, you need to add a second option:

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

## Strategies provided by Hypothesis

Here is a selection of strategies provided by Hypothesis that may be useful to know:

- `integers()`. Generates integers.
- `floats()`. Generates floats.
- `booleans()`. Generates booleans.
- `text()`. Generates unicode strings (i.e., instances of `str`).
- `lists()`. Generates lists with elements from the strategy passed to it.

  For instance, `st.lists(st.integers())` generates lists of integers.
- `tuples()`. Generates tuples of a fixed length.

  For instance, `st.tuples(st.integers(), st.floats())` generates tuples with two elements, where the first element is an integer and the second is a float.
- `one_of()`. Generates from any of the strategies passed to it.

  For instance, `st.one_of(st.integers(), st.floats())` generates either integers or floats. You can also use `|` to construct `one_of()`, like `st.integers() | st.floats()`.
- `builds()`. Generates instances of a class (or other callable) by specifying a strategy for each argument.

  For instance: `st.builds(Person, name=st.text(), age=st.integers())`.
- `just()`. Generates the exact value passed to it: `st.just("a")` generates the exact string `"a"`. This is useful when something expects to be passed a strategy.

  For instance, `st.lists(st.integers() | st.just("a"))` generates lists whose elements are either integers or the string `"a"`.
- `sampled_from()`. Generates a random value from a list. For instance, `st.sampled_from(["a", 1, True])` is equivalent to:

  `st.just("a") | st.just(1) | st.just(True)`.
- `none()`. Generates `None`.

  Useful for parameters that can be optional, like `st.integers() | st.none()`.