**Computer Systems Modeling and Verification**
**(USEEN1)**

# Part II: Design-by-contract

Deal is a Python library for design-by-contract. The following tutorial is taken from Deal documentation, section Contract-Driven Development.

## Tutorial

Let's take for example an incredibly simple code and imagine that it's incredibly complicated logic.

```
def cat(left: str, right: str) -> str:
    """␣Concatenate␣two␣given␣strings.
    """
    return left + right
```

## Tests

How can we be sure this code works? No, it's not obvious. Remember the rules of the game, we have an incredibly complicated realization. So, we can't say it works or not while we haven't tested it.

```
def test_cat() -> None:
    result = cat('abc', 'def')
    assert result == 'abcdef'
```

Now, run pytest:

```
pytest cat.py
```

It passes. So, our code works. Right?

## Table tests

Wait, but what about corner cases? What if one string is empty? What if both strings are empty? What if we have only one character in both strings? We need check more values and this is where table driven tests will save our time. In pytest, we can use @pytest.mark.parametrize to make such tables.

```
import pytest

@pytest.mark.parametrize(("left", "right", "expected"), [
    ('a', 'b', 'ab'),
    ('', '', ''),
    ('', 'b', 'b'),
    ('a', '', 'a'),
    ('text', 'check', 'textcheck'),
])
def test_cat(left: str, right: str, expected: str) -> None:
    result = cat(left, right)
    assert result == expected
```

## Properties

Table tests can be enormously long, and for every test case, we have to manually calculate the expected result. For complicated code, it's a lot of work. Can we do it better and think and write less? Yes, we can instead of *expected result* talk about *expected properties of the result*. The big difference is the result is different for different input values, but properties always the same. The coolest thing is in most cases you already know result properties, it is the business requirements, and your code is no more than the implementation of these requirements.

So, what are the properties of our function?

1. The result string starts with the first given string.
2. The result string ends with the second given string.
3. Result string has the length equal to the sum of lengths of given strings.

Now, we can check these properties for the result instead of checking particular values.

```
@pytest.mark.parametrize(("left", "right"), [
    ('a', 'b'),
    ('', ''),
    ('', 'b'),
    ('a', ''),
    ('text', 'check'),
])
def test_cat(left, right):
    result = cat(left=left, right=right)
    assert result.startswith(left)
    assert result.endswith(right)
    assert len(result) == len(left) + len(right)
```

## Exceptions

With pytest, you can also test that a function call raises some exception. For instance, you can test that a division by zero actually raises `ZeroDivisionError` as follows:

```
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

## Exercises

For each of the functions from the assignment on "recursion on lists":

- test the functional implementation using properties (and parametrized tests)
- provide an imperative implementation of the function and test it using the same tests
- test again the imperative version using the functional version as its specification