TESTS FÜR FAULE

EINFÜHRUNG IN PROPERTY-BASED TESTING

Michael Seifert

WAS IST EIN PROPERTY-BASED-TEST?

```python
from typing import List, TypeVar

T = TypeVar("T", int, float)

def max(l: List[T]) -> T:
    current_max = None
    for element in l:
        if current_max is None or element > current_max:
            current_max = element
    return current_max
```

```python
def test_max_returns_maximum_int():
    values = [-3, 5, 1]
    assert max(values) == 5

def test_max_returns_maximum_float():
    values = [-3.0, 5.0, 1.0]
    assert max(values) == 5.0
```

```python
@pytest.mark.parametrize(
    "values, expected_max",
    (
        ([-3, 5, 1], 5),
        ([-3.0, 5.0, 1.0], 5.0),
    )
)
def test_max_returns_max(values, expected_max):
    assert max(values) == expected_max
```

REICHEN ZWEI TESTS AUS?

```
@pytest.mark.parametrize(
    "values, expected_max",
    (
        ([-3, 5, 1], 5),
        ([-3.0, 5.0, 1.0], 5.0),
    )
)
def test_max_returns_max(values, expected_max):
    assert max(values) == expected_max
```

```python
@pytest.mark.parametrize(
    "values",
    (
        [-3, 5, 1],
        [-3.0, 5.0, 1.0],
    )
)
def test_max_returns_max(values):
    assert max(values) == sorted(values)[-1]
```

```python
@given(
    st.one_of(
        st.lists(st.integers()),
        st.lists(st.floats()),
    )
)
def test_max_returns_max(values):
    assert max(values) == sorted(values)[-1]
```

```
@given(
    st.one_of(
        st.lists(st.integers()),
        st.lists(st.floats()),
    )
)
def test_max_returns_max(values):
    assert max(values) == sorted(values)[-1]
```

```
    def test_max_returns_max(values):
>       assert max(values) == sorted(values)[-1]
E       IndexError: list index out of range
```

```python
from typing import List, TypeVar

T = TypeVar("T", int, float)

def max(l: List[T]) -> T:
    current_max = None
    for element in l:
        if current_max is None or element > current_max:
            current_max = element
    return current_max
```

```python
from typing import List, TypeVar

T = TypeVar("T", int, float)

def max(l: List[T]) -> T:
    if not l:
        raise ValueError()
    current_max = None
    for element in l:
        if current_max is None or element > current_max:
            current_max = element
    return current_max
```

```python
from typing import List, TypeVar

T = TypeVar("T", int, float)

def max(l: List[T]) -> T:
    if not l:
        raise ValueError()
    current_max = None
    for element in l:
        if current_max is None or element > current_max:
            current_max = element
    return current_max
```

```python
def test_max_raises_when_input_is_empty():
    with pytest.raises(ValueError):
        max([])
```

TAKE AWAYS

TAKE AWAYS

- Als Autor von Source Code schreibt man im Allgemeinen keine Tests, die Fehler aufdecken.

TAKE AWAYS

- Als Autor von Source Code schreibt man im Allgemeinen keine Tests, die Fehler aufdecken.
- Property-Based Testing generiert mehrere Testfälle, die eine Eigenschaft der Software überprüft.

WARUM PROPERTY-BASED TESTING?

```
def square(i: int) -> int:
    return i*i
```

```
def test_square_positive_number():
    number = 42
    result = square(number)
    assert result == 1764
```

```
def test_square_positive_number():
    number = 42
    result = square(number)
    assert result == 1764
```

```
def test_square_negative_number():
    number = -5
    result = square(number)
    assert result == 25
```

```
def test_square_positive_number():
    number = 42
    result = square(number)
    assert result == 1764
```

```
def test_square_negative_number():
    number = -5
    result = square(number)
    assert result == 25
```

```
def test_square_zero():
    number = 0
    result = square(number)
    assert result == 0
```

```python
@given(st.integers())
def test_square(number: int):
    assert square(number) == number**2
```

```
0
120
0
-5476758948831415126
0
-27053
-105
9066180397792449019
-60
118
-25198
-5425
-56
-1
```

WARUM PROPERTY-BASED TESTING?

Verbesserte Wartbarkeit

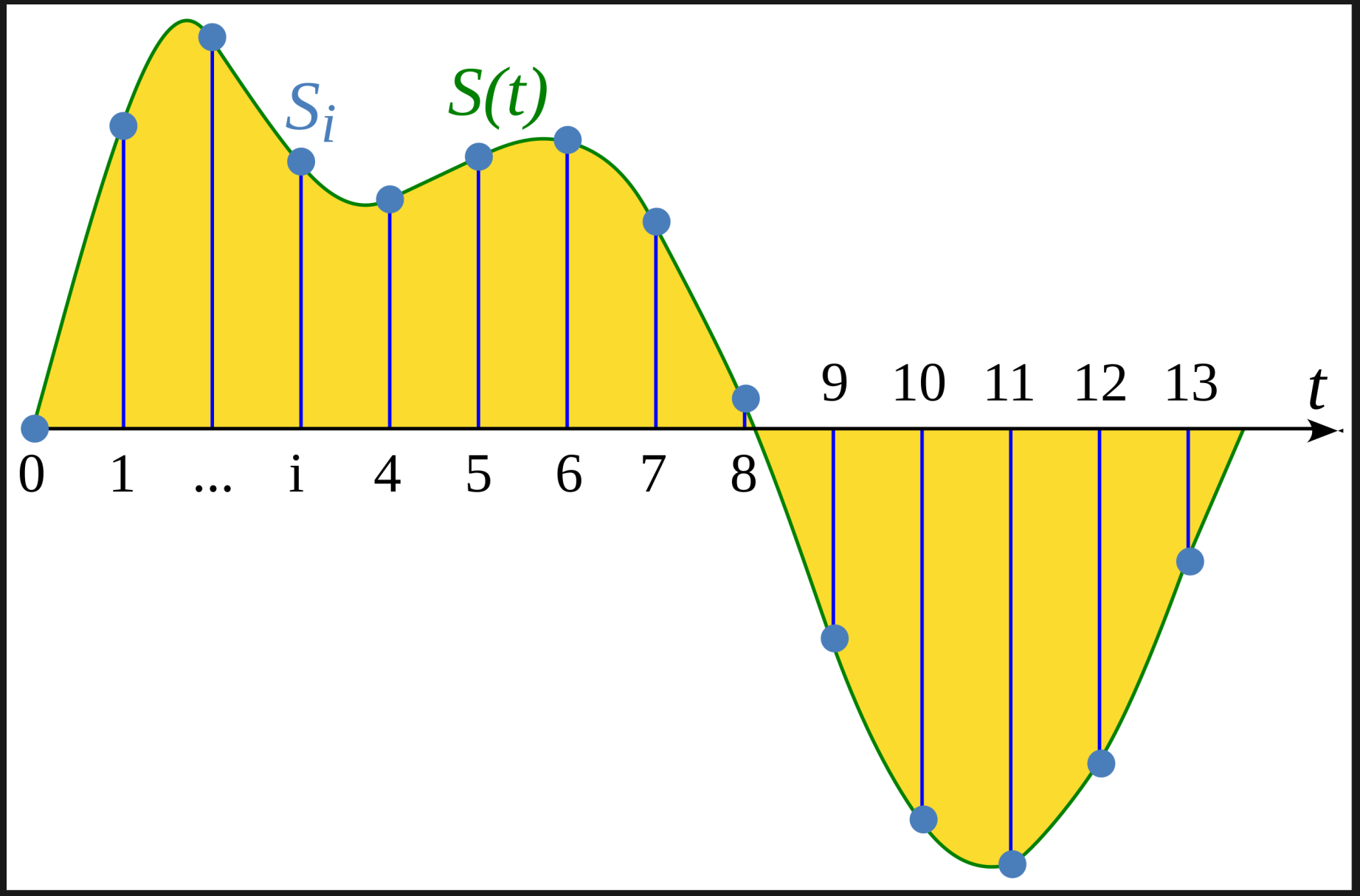
WARUM PROPERTY-BASED TESTING?

Verbesserte Wartbarkeit

Potentiell bessere Testabdeckung

WARUM PROPERTY-BASED TESTING?

WARUM PROPERTY-BASED TESTING?


S_i
$S(t)$
9
10
11
12
13
t
0
1
...
i
4
5
6
7
8

```
class Waveform:
    samples: list[int]

    def amplitude(self) -> int:
        ...
```

```python
class Waveform:
    samples: list[int]

    def amplitude(self) -> int:
        ...
```

```python
def amplify_loudness(
    audio_signal: Waveform,
    amplification: float,
) -> Waveform:
    ...
```

```
def test_amplify_loudness_increases_amplitude(
    signal: Waveform,
    amplification: float
):
    amplified = amplify_loudness(waveform, amplification)

    assert amplified.amplitude > signal.amplitude
```

```
def test_amplify_loudness_increases_amplitude(
    signal: Waveform,
    amplification: float
):
    amplified = amplify_loudness(waveform, amplification)

    assert amplified.amplitude > signal.amplitude
```

```
def test_amplify_loudness_does_not_modify_sample_count(
    signal: Waveform,
    amplification: float
):
    amplified = amplify_loudness(waveform, amplification)

    assert len(amplified.samples) == len(signal.samples)
```

Property-Based Testing erlaubt das Testen von Funktionen ohne Kenntnis über das exakte Ergebnis

HERANGEHENSWEISE

FUZZING

```
@given(
    st.lists(st.integers())
    | st.lists(st.floats())
    | st.lists(st.text())
)
def test_sort(a_list):
    custom_sort(a_list)
```

DIFFERENZIELLE TESTS

```
@given(
    st.lists(
        st.text()
    )
)
def test_my_custom_sort(l):
    assert custom_sort(l) == sorted(l)
```

ROUNDTRIPS

```
@given(st.binary())
def test_base64(binary):
    encoded = b64encode(binary)
    decoded = b64decode(encoded)
    assert decoded == binary
```

METAMORPHE TESTS

```
@given(st.integers(min_value=0))
def test_negative_square_equals_square(n):
    assert square(n) == square(-n)
```

METAMORPHE TESTS

```
@given(st.integers(min_value=0))
def test_negative_square_equals_square(n):
    assert square(n) == square(-n)
```

```
@given(st.integers(min_value=0))
def test_square_is_strictly_monotonic(n):
    assert square(n) < square(n + 1)
```

ALGEBRAISCHE EIGENSCHAFTEN

```
@given(st.integers())
def test_sign_is_idempotent(n):
    assert sign(n) == sign(sign(n))
```

Fuzzing

Differenzielle Tests

Roundtrips

Metamorphe Tests

Algebraische Eigenschaften

WARUM KEIN PROPERTY-BASED TESTING?

WARUM KEIN PROPERTY-BASED TESTING?

Erhöht Laufzeit von Tests

LIBRARIES

Python: Hypothesis

JavaScript: JSVerify

TypeScript: fast-check

Java: junit-quickcheck

There are two ways to write error-free programs; only the third one works.

—Alan J. Perlis, Epigrams in Programming (1982)

There are two ways to write error-free programs; only the third one works.

—Alan J. Perlis, Epigrams in Programming (1982)

www.seifertm.de

LinkedIn, GitHub: @seifertm

Twitter: @seifertm0