

# Property based testing with Hypothesis



The-Compiler/hypothesis-talk

Florian Bruhin

October 17th, 2024



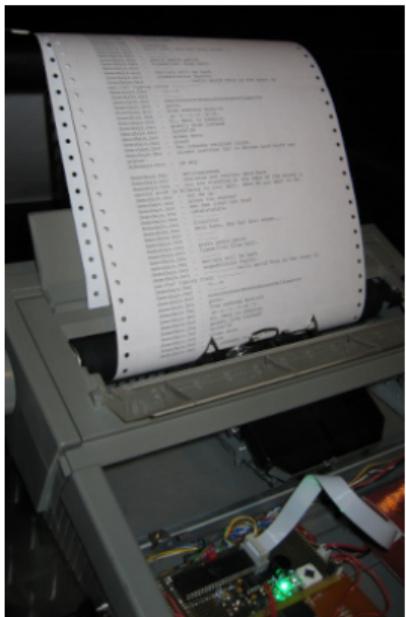
BRUHIN  
SOFTWARE



## About me



The-Compiler/hypothesis-talk

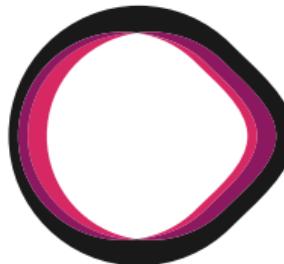


2011



pytest

2013, 2015



BRUHIN  
SOFTWARE

2019, 2020

Here be dragons



The-Compiler/  
hypothesis-talk

## Calculator project

— rpnCalc/utils.py —————

```
def calc(a, b, op):
    if op == "+":
        return a + b
    elif op == "-":
        return a - b
    elif op == "*":
        return a * b
    elif op == "/":
        return a / b
    raise ValueError("Invalid operator")
```

# Reverse Polish Notation (RPN)

## History



- Using a calculator without needing a = key, and without parentheses
- Makes it much easier to implement, using a stack data structure
- Used by all HP calculators in the 1970s–80s, still used by some today
- Displayed here: HP 12C financial calculator, introduced in 1981, still in production today (HP's longest and best-selling product)

# Reverse Polish Notation (RPN)

## Explanation

$$1 + 2$$

1    2

↶    ↶

+

2  
1

2  
1

3

# Reverse Polish Notation (RPN)

## Explanation

$$5 \cdot (1 + 2)$$

1    2

↶    ↶

+

5

↶

\*

2  
1

2  
1

5  
3

5  
3

15

## Reverse Polish Notation (RPN)

— rpncalc/rpn\_v1.py —————

```
from rpncalc.utils import calc

class RPNCcalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)
```

## Reverse Polish Notation (RPN)

— rpncalc/rpn\_v1.py —————

```
from rpncalc.utils import calc

class RPNCcalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)
```

# Reverse Polish Notation (RPN)

— rpnCalc/rpn\_v1.py —

```
from rpnCalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        if inp.isdigit():
            n = float(inp)
            self.stack.append(n)
        elif inp in "+-*/":
            b = self.stack.pop()
            a = self.stack.pop()
            res = calc(a, b, inp)
            self.stack.append(res)
            print(res)
```

```
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/":
        b = self.stack.pop()
        a = self.stack.pop()
        res = calc(a, b, inp)
        self.stack.append(res)
        print(res)

if __name__ == "__main__":
    rpn = RPNCalculator()
    rpn.run()
```

# Reverse Polish Notation (RPN)

— rpncalc/rpn\_v1.py —

```
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        if inp.isdigit():
            n = float(inp)
            self.stack.append(n)
        elif inp in "+-*/":
            b = self.stack.pop()
            a = self.stack.pop()
            res = calc(a, b, inp)
            self.stack.append(res)
            print(res)
```

```
if __name__ == "__main__":
    rpn = RPNCalculator()
    rpn.run()
```

# Reverse Polish Notation (RPN)

— rpncalc/rpn\_v1.py —

```
from rpncalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        if inp.isdigit():
            n = float(inp)
            self.stack.append(n)
        elif inp in "+-*/":
            b = self.stack.pop()
            a = self.stack.pop()
            res = calc(a, b, inp)
            self.stack.append(res)
            print(res)
```

```
if __name__ == "__main__":
    rpn = RPNCalculator()
    rpn.run()
```

# Reverse Polish Notation (RPN)

— rpnCalc/rpn\_v1.py —

```
from rpnCalc.utils import calc

class RPNCalculator:
    def __init__(self) -> None:
        self.stack = []

    def run(self) -> None:
        while True:
            inp = input("> ")
            if inp == "q":
                return
            elif inp == "p":
                print(self.stack)
            else:
                self.evaluate(inp)

    def evaluate(self, inp: str):
        if inp.isdigit():
            n = float(inp)
            self.stack.append(n)
        elif inp in "+-*/":
            b = self.stack.pop()
            a = self.stack.pop()
            res = calc(a, b, inp)
            self.stack.append(res)
            print(res)
```

```
if __name__ == "__main__":
    rpn = RPNCalculator()
    rpn.run()
```

# Reverse Polish Notation (RPN)

Towards an improved version

- Allow negative numbers and floating-point inputs, not just `.isdigit()`
- Fix `+-` being treated as valid input due to `elif inp in "+-*/"`:
- Print error when using an invalid operator
- Handle `ZeroDivisionError` when dividing by zero,  
and `IndexError` with < 2 elements on stack

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpn calc/rpn\_v1.py —

```
...
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/":
        ...
...
```

[mathspp.com/blog/pydonts/  
eafp-and-lbyl-coding-styles](https://mathspp.com/blog/pydonts/eafp-and-lbyl-coding-styles)

**LBYL:** Look before you leap

— rpn calc/rpn\_v2.py —

```
...
def evaluate(self, inp: str) -> None:
    try:
        self.stack.append(float(inp))
    return
except ValueError:
    pass
```

if inp not in ["+", "-", "\*", "/"]:

**EAFP:** It's easier to ask for forgiveness,  
than for permission

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpncalc/rpn\_v1.py —————

```
...
def evaluate(self, inp: str):
    if inp.isdigit():
        n = float(inp)
        self.stack.append(n)
    elif inp in "+-*/":
        ...
    else:
```

— rpncalc/rpn\_v2.py —————

```
...
def evaluate(self, inp: str) -> None:
    try:
        self.stack.append(float(inp))
    return
except ValueError:
    pass
```

```
if inp not in ["+", "-", "*", "/"]:
    ...
else:
```

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpnCalc/rpn\_v2.py —

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return
```

```
b = self.stack.pop()
a = self.stack.pop()

try:
    res = calc(a, b, inp)
except ZeroDivisionError:
    self.err("Division by zero")
    return

self.stack.append(res)
print(res)
```

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpnCalc/rpn\_v2.py —

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return
```

```
b = self.stack.pop()
a = self.stack.pop()

try:
    res = calc(a, b, inp)
except ZeroDivisionError:
    self.err("Division by zero")
    return

self.stack.append(res)
print(res)
```

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpnCalc/rpn\_v2.py —

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return
```

```
b = self.stack.pop()
a = self.stack.pop()

try:
    res = calc(a, b, inp)
except ZeroDivisionError:
    self.err("Division by zero")
    return

self.stack.append(res)
print(res)
```

# Reverse Polish Notation (RPN)

Code: Fixing bugs, improving error handling

— rpnCalc/rpn\_v2.py —

```
def err(self, msg: str) -> None:
    print(msg, file=sys.stderr)

def evaluate(self, inp: str) -> None:
    ...
    if inp not in ["+", "-", "*", "/"]:
        self.err(
            f"Invalid input: {inp}")
        return

    if len(self.stack) < 2:
        self.err("Not enough operands")
        return
```

```
b = self.stack.pop()
a = self.stack.pop()

try:
    res = calc(a, b, inp)
except ZeroDivisionError:
    self.err("Division by zero")
    return

self.stack.append(res)
print(res)
```



# Property-based testing with Hypothesis

## Motivation

From the Hypothesis website, [hypothesis.works](https://hypothesis.works):

*Most testing is ineffective*

*Normal “automated” software testing is surprisingly manual.*

*Every scenario the computer runs, someone had to write by hand.*

*Hypothesis can fix this.*

# Property-based testing with Hypothesis

## Motivation

From the Hypothesis website, [hypothesis.works](https://hypothesis.works):

*Most testing is ineffective*

*Normal “automated” software testing is surprisingly manual.*

*Every scenario the computer runs, someone had to write by hand.*

*Hypothesis can fix this.*

Idea behind “Property-Based testing” (popularized by QuickCheck/Haskell):

- Generate input data based on a *strategy* (how should the data look?)
- Run the test case many times (say, 100), with different random generated data
- Check for *properties* / *invariants* that should hold for all input data
- If there is a failure, *minimize* the input data to a minimal failing example

# Property-based testing with Hypothesis

Old buggy RPN calculator

```
— rpncalc/rpn_v1.py —————
```

```
class RPNCcalculator:  
    ...  
  
    def evaluate(self, inp: str):  
        if inp.isdigit():  
            n = float(inp)  
            self.stack.append(n)  
        elif inp in "+-*/":  
            b = self.stack.pop()  
            a = self.stack.pop()  
            res = calc(a, b, inp)  
            self.stack.append(res)  
        print(res)
```

```
> 1
```

```
> +
```

```
Traceback (most recent call last):
```

```
...  
File ..., in <module>
```

```
    rpn.run()
```

```
File ..., in run
```

```
    self.evaluate(inp)
```

```
File ..., in evaluate
```

```
    a = self.stack.pop()
```

```
IndexError: pop from empty list
```

# Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

```
— tests/test_hypothesis_rpncalc_v1.py
```

```
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCcalculator
```

```
@given(st.text())
def test_random_strings(s: str):
    rpn = RPNCcalculator()
    rpn.evaluate(s)
```

# Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

```
— tests/test_hypothesis_rpncalc_v1.py
```

```
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCcalculator
```

```
@given(st.text())
def test_random_strings(s: str):
    rpn = RPNCcalculator()
    rpn.evaluate(s)
```

```
>     b = self.stack.pop()
E   IndexError: pop from empty list
E   Falsifying example:
E       test_random_strings(s='')
```

# Property-based testing with Hypothesis

Old buggy RPN calculator: Hypothesis test

```
— tests/test_hypothesis_rpncalc_v1.py
```

```
from hypothesis import given, strategies as st
from rpncalc.rpn_v1 import RPNCcalculator
```

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

```
def test_random_strings(s: str):
```

```
    rpn = RPNCcalculator()
```

```
    rpn.evaluate(s)
```

```
>     b = self.stack.pop()
```

```
E     IndexError: pop from empty list
```

```
E     Falsifying example:
```

```
E         test_random_strings(s='')
```

```
elif inp in "+-*/":
```

```
    ...
```

```
>>> "" in "+-*/"
```

```
True
```

# Property-based testing with Hypothesis

Surely v2 is bug-free?

```
— rpncalc/rpn_v2.py —————
```

```
class RPNCcalculator:  
    ...  
  
    def evaluate(self, inp: str) -> None:  
        ...  
        if len(self.stack) < 2:  
            print("Not enough operands")  
            return  
  
        b = self.stack.pop()  
        a = self.stack.pop()
```

# Property-based testing with Hypothesis

Surely v2 is bug-free?

```
— rpncalc/rpn_v2.py —————
```

```
class RPNCcalculator:  
    ...  
  
    def evaluate(self, inp: str) -> None:  
        ...  
        if len(self.stack) < 2:  
            print("Not enough operands")  
            return  
  
        b = self.stack.pop()  
        a = self.stack.pop()
```

test\_random\_strings PASSED

# Property-based testing with Hypothesis

Surely v2 is bug-free?

— tests/test\_hypothesis\_rpncalc\_v2.py

```
@given(st.integers(), st.integers())
def test_operators(n1: int, n2: int):
    rpn = RPNCcalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
```

# Property-based testing with Hypothesis

Surely v2 is bug-free?

```
— tests/test_hypothesis_rpncalc_v2.py
```

```
@given(st.integers(), st.integers())
def test_operators(n1: int, n2: int):
    rpn = RPNCcalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E    assert [1.055531162664984 8 e+16] == [1055531162664984 9 ]
E    ...
E    Falsifying example: test_operators(
E        n1=0,
E        n2=10555311626649849 ,
E    )
```

# Property-based testing with Hypothesis

Surely v2 is bug-free?

```
— tests/test_hypothesis_rpncalc_v2.py

@given(st.integers(), st.integers())
def test_operators(n1: int, n2: int):
    rpn = RPNCcalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E    assert [1.0555311626649848e+16] == [10555311626649849]
E
E    ...
E    Falsifying example: test_operators(
E        n1=0,
E        n2=10555311626649849,
E    )
```

# Property-based testing with Hypothesis

Surely v2 is bug-free?

```
— tests/test_hypothesis_rpncalc_v2.py
```

```
@given(st.integers(), st.integers())
def test_operators(n1: int, n2: int):
    rpn = RPNCcalculator(Config())
    rpn.evaluate(str(n1))
    rpn.evaluate(str(n2))
    rpn.evaluate("+")
    assert rpn.stack == [n1 + n2]
E    assert [1.0555311626649848e+16] == [10555311626649849]
E
E    ...
E    Falsifying example: test_operators(
E        n1=0,
E        n2=10555311626649849,
E    )
```

---

```
— rpncalc/rpn_v2.py —————
...
self.stack.append(float(inp))
```

# Property-based testing with Hypothesis

More sophisticated strategies

```
@given(  
    st.lists(  
        st.one_of(  
            st.integers().map(str),  
            st.floats().map(str),  
            st.just("+"), st.just("-"),  
            st.just("*"), st.just("/"),  
        )  
    )  
)  
  
def test_usage(inputs: list[str]):  
    rpn = RPNCcalculator(Config())  
    for inp in inputs:  
        rpn.evaluate(inp)
```

# Property-based testing with Hypothesis

More sophisticated strategies

```
@given(  
    st.lists(  
        st.one_of(  
            st.integers().map(str),  
            st.floats().map(str),  
            st.just("+"), st.just("-"),  
            st.just("*"), st.just("//"),  
        )  
    )  
  
def test_usage(inputs: list[str]):  
    rpn = RPNCcalculator(Config())  
    for inp in inputs:  
        rpn.evaluate(inp)
```

Trying example: test\_usage(  
inputs=[  
 '-2.2250738585e-313',  
 '/', '-10', '110', '+', '+'])  
Not enough operands  
100.0  
100.0

# Property-based testing with Hypothesis

More sophisticated strategies

```
@given(  
    st.lists(  
        st.one_of(  
            st.integers().map(str),  
            st.floats().map(str),  
            st.just("+"), st.just("-"),  
            st.just("*"), st.just("//"),  
        )  
    )  
  
def test_usage(inputs: list[str]):  
    rpn = RPNCcalculator(Config())  
    for inp in inputs:  
        rpn.evaluate(inp)
```

Trying example: test\_usage(  
inputs=[  
 '-2.2250738585e-313',  
 '/', '-10', '110', '+', '+'])

Not enough operands  
100.0  
100.0

Trying example: test\_usage(  
inputs=['+', '+', '-', '\*'],  
)

Not enough operands  
Not enough operands  
Not enough operands  
Not enough operands

# Property-based testing with Hypothesis

Another example: Run length encoding

Original:

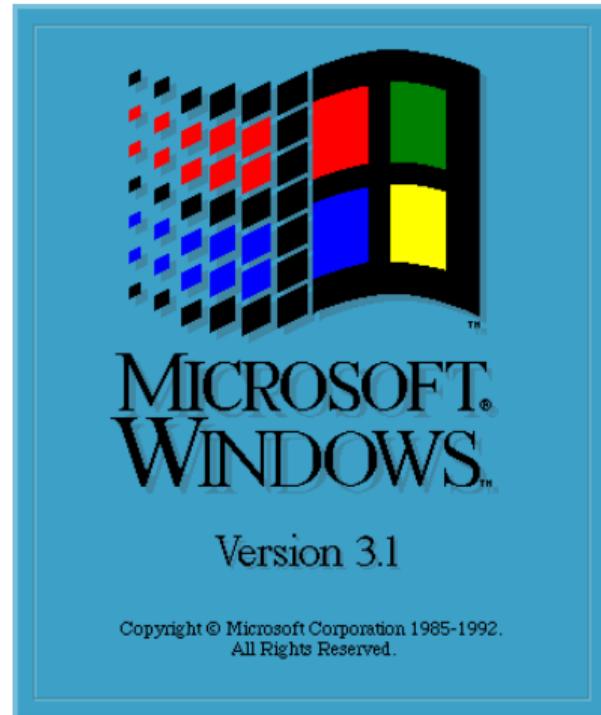
PPPPPYYYPPPPPPPYYYY

Compressed:

5P3Y7P4Y

Used in:

- Fax machines
- Windows 3.1 startup screen
- ...



# Property-based testing with Hypothesis

## Decoding

```
input: [(5, "P"), (3, "Y"), (7, "P"), (4, "Y")]
output: "PPPPPYYYYPPPPPPPYYYY"
```

---

```
— tests/test_hypothesis.py —
```

```
def decode(lst: list[tuple[int, str]]) -> str:
    s = ""
    for count, character in lst:
        s += count * character
    return s
```

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:

"PPPPPYYYYYYYYYYYY"

output:

```
[  
    (5, "P"),  
    (3, "Y"),  
    (7, "P"),  
    (4, "Y"),  
]
```

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:

"PPPPPYYYPPPPPPPYYYY"

output:

```
[  
    (5, "P"),  
    (3, "Y"),  
    (7, "P"),  
    (4, "Y"),  
]
```

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:

"PPPPPYYYPPPPPPPYYYY"

output:

```
[  
    (5, "P"),  
    (3, "Y"),  
    (7, "P"),  
    (4, "Y"),  
]
```

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:

"PPPPPYYYPPPPPPPYYYY"

output:

```
[  
    (5, "P"),  
    (3, "Y"),  
    (7, "P"),  
    (4, "Y"),  
]
```

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:  
"PPPPPYYYPPPPPPPYYYY"

output:  
[  
 (5, "P"),  
 (3, "Y"),  
 (7, "P"),  
 (4, "Y"),  
]

# Property-based testing with Hypothesis

## Encoding

```
def encode(input_string: str) -> list[tuple[int, str]]:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        if character != prev:  
            if prev:  
                entry = (count, prev)  
                lst.append(entry)  
            count = 1  
            prev = character  
        else:  
            count += 1  
    entry = (count, character)  
    lst.append(entry)  
    return lst
```

input:  
"PPPPPYYYPPPPPPPYYYY"

output:  
[  
 (5, "P"),  
 (3, "Y"),  
 (7, "P"),  
 (4, "Y"),  
]

# Property-based testing with Hypothesis

## Manual test

```
@pytest.mark.parametrize("inp, out", [
    ("PYY", [(1, "P"), (2, "Y")]),
    ...
])
def test_rle(inp: str, out: list[tuple[int, str]]):
    assert encode(inp) == out
```

Can we think of all corner-cases?

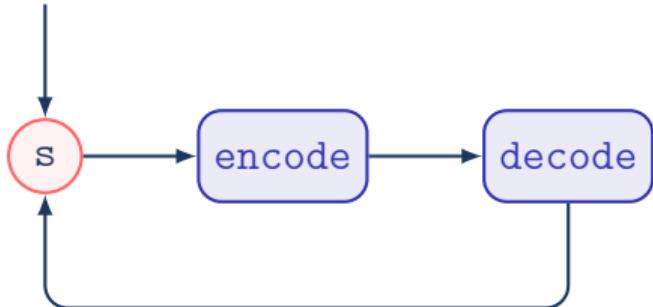
Maybe it's easier to think of *invariants* (certain properties that always hold true)?

# Property-based testing with Hypothesis

## Test with Hypothesis

```
@given(text())
def test_decode_inverts_encode(s: str):
    assert decode(encode(s)) == s
```

```
@given(text())
```



# Property-based testing with Hypothesis

## Result

```
UnboundLocalError:  
local variable 'character'  
referenced before assignment
```

```
Falsifying example:  
test_decode_inverts_encode(s='')
```

```
def encode(input_string: str) -> str:  
    count = 1; prev = ""; lst = []  
    for character in input_string:  
        ...  
        entry = (count, character)  
        lst.append(entry)  
    return lst
```

# Property-based testing with Hypothesis

**Demo** `plugins/test_hypothesis.py`:

- Fix the issue, e.g. by adding `if not input_string: return []` at the beginning of the encode-function.
- Re-run the test, it should pass.  
Run with `--hypothesis-verbosity=verbose` and `-s` to observe examples.
- Deliberately introduce a more subtle issue, e.g. break resetting the count by commenting out the second `count = 1`.
- Re-run the test – you should get a minimal example which fails the test.

# Property-based testing with Hypothesis

Another possibility

```
def encode_itertools(input_string: str) -> list[tuple[int, str]]:  
    return [  
        (len(list(group)), character)  
        for character, group  
        in itertools.groupby(input_string)  
    ]  
  
@given(text())  
def test_alternative_implementation(s: str):  
    assert encode_itertools(s) == encode(s)
```

# Property-based testing with Hypothesis

## What it can do

- Generate data based on types/strategies
- Combine/filter various strategies to generate more sophisticated data
- Generate names, etc. via fakefactory
- Generate data matching a Django model
- Generate data matching a grammar
- Generate test code (rather than data) based on a state machine

# Upcoming events

- **March 4th to 6th, 2025**

Python Academy

([python-academy.com](http://python-academy.com)):

Professional Testing with Python  
Leipzig, Germany & Remote

- **Custom training / coaching:**

- Python
- pytest
- GUI programming with Qt
- Best Practices  
(packaging, linting, etc.)
- Git
- ...

Remote or on-site

[florian@bruhiin.software](mailto:florian@bruhiin.software)

<https://bruhiin.software/>



**BRUHIN**  
SOFTWARE



# Feedback and questions



**Florian Bruhin**



florian@bruhiin.software



bruhiin.software



@the\_compiler



@The-Compiler



linkedin.com/in/florian-bruhin



**BRUHIN**  
SOFTWARE

Copyright 2015 – 2024 Florian Bruhin



CC BY 4.0

