CSC Lecture 8: Porperty-Based Testing and Nested Quantifiers

Hisbaan Noorani

September 29, 2020

Contents

1 Ex 1: Property-basted testing

1. Consider the `divides` function from lecture.

```python
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
    possible_divisors = range(- abs(n), abs(n) + 1)
    return any({n == k * d for k in possible_divisors})
```

There are many different properties of divisibility from mathematics that we can use to express property-based tests. For each of the properties below, translate them into a property-based test. We've started the first one for you (you can copy-and-paste the template and use it for each one; you don't need to repeat the import statements). You don't need to include doctests.

(a) I$\forall a, d \in \mathbb{Z}, d|ad$

```python
from hypothesis import given
from hypothesis.strategies import integers


@given(a=integers(), d=integers())
def test_a(a: int, d: int) -> None:
    assert divides(d, d * a)
```

(b) $\forall n \in \mathbb{Z}, 1|n$

```python
from hypothesis import given
from hypothesis.strategies import integers


@given(n=integers())
def test_a(n: int) -> None:
    assert divides(1, n)
```

(c) $\forall n \in \mathbb{Z}, n|n$

```python
from hypothesis import given
from hypothesis.strategies import integers


@given(n=integers())
def test_a(n: int) -> None:
    assert divides(n, n)
```

(d) $\forall d, n \in \mathbb{Z}, d|n \Rightarrow d|n+d$

```python
from hypothesis import given
from hypothesis.strategies import integers


@given(d=integers(), n=integers())
def test_a(d: int, n: int) -> None:
    assert not divides(d, n) or divides(d, n + d)
```

2 Ex 2: Nested Quantifiers

Consider the following table that shows the employees of a (very small) company:

Employee	Salary	Department
Aizah	70,000	Sales
Betty	25,000	Sales
Carlos	50,000	HR
Doug	40,000	Sales
Ellen	60,000	Design
Flo	30,000	Design

Let E be the set of six employees listed above. We'll define two predicates on this set:

$$Rich(x) : x \text{ earns more than 45,000, where } x \in E$$
$$SameDept(x, y) : x \text{ and } y \text{ are in the same department, where } x, y \in E$$

1. Consider the statement

$$\exists x, y \in E, Rich(x) \wedge SameDept(x, y)$$

Give one example to show that this statement is True. Is there more than one possible answer? Note: just as in programming, the two variables x and y can have the samve value (i.e. refer to the same employee).

Let x = Aizah.

Let y = Betty.

Aizah is rich.

They are both in sales.

Yes, there is more than one possible answer.

2. Here's the same statement, but with a universal quantifier instead:

$$\forall x, y \in E, Rich(x) \wedge SameDept(x, y)$$

Give one counterexample to show that this statement is False. Is there more than one possible answer?

Let x = Flo.

Let y = Ellen.

Flo is not rich.

Ellen and flow are in the same department but that does not matter because Flo is not rich.

Yes, there is more than one possible answer.

3. Now let's look at alternating quantifiers.

$$\forall y \in E, \exists x \in E, Rich(x) \wedge SameDept(x, y)$$

Is this True or False? How do you know?

True, there at least one rich employee in every department.

4. With the quantifiers switched:

$$\exists x \in E, \forall y \in E, Rich(x) \wedge SameDept(x, y)$$

Explain why this statement is False.

This statment claims that all people in the set E are in the same department. This is not true.

3 Ex 3: Nested quantifiers in Python

Suppose we have the following definitions in Python:

- A variable `employees` referring to a set of values representing employees (Don't worry about the exact data type here.)
- A function `is_rich` that takes an employee and returns a `bool` representing whether that employee earns more than 45,000.
- A functoin `same_dept` that takes two employees and returns whether they are in the same department.
- Given these Python variables and functions, write expressions to express each of the four statements from the previous exercise.

  Note that for the alternating quantifiers, you'll need to use nested `and/all` calls. If you have time, try writing a separate function for the inner `any/all` call to make your code easier to understand.

```
EMPLOYEES = [
    ('Aizah', 70000, 'Sales'),
    ('Betty', 25000, 'Sales'),
    ('Carlos', 50000, 'HR'),
    ('Doug', 40000, 'Sales'),
    ('Ellen', 60000, 'Design'),
    ('Flo', 30000, 'Design')
]

def is_rich(employee: tuple) -> bool:
    return employee[1] > 45000

def same_dept(employee1: tuple, employee2: tuple) -> bool:
```

```
    return employee1[2] == employee2[2]

one = any({is_rich(x) and same_dept(x, y) for x in EMPLOYEES for y in EMPLOYEES})

two = all({is_rich(x) and same_dept(x, y) for x in EMPLOYEES for y in EMPLOYEES})

three = all({any({is_rich(x) and same_dept(x, y) for x in EMPLOYEES} for y in EMPLOYEES)})

four = any({all({is_rich(x) and same_dept(x, y) for x in EMPLOYEES} for y in EMPLOYEES)})
```