CSC110 Lecture 8: Function Specification and Property-Based Testing

David Liu, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

Announcements & Today’s Plan

Announcements

- Assignment 1 has been posted!
  - Check the FAQ (+ corrections) page
  - Additional TA office hours (schedule on Quercus)
  - **(NEW)** Academic Integrity in CSC110 advice page
  - **(NEW)** Review the Assignment Policies page, including grace credit policy
- Join a Recognized Study Group
- **(NEW)** Term Test 1 info has been posted!
  - Info on Quercus:
    - Test time and location (not MY 150!)
    - Test coverage
    - Advice for advice
  - Review provided reference sheet

Today you'll learn to...

1. Define the terms precondition, postcondition, and function correctness.
2. Write detailed type annotations for collections in Python.
3. Document function preconditions as Python expressions in function docstrings.
4. Use PythonTA to check function preconditions and postconditions automatically.
5. Write property-based tests to check correctness of Python functions.
6. Differentiate between unit tests and property-based tests.

Prep Recap: Function Correctness

Function specification: preconditions and postconditions

Function **precondition**: a predicate that the function's inputs must satsify.

Function **postcondtion**: a predicate that the functions return value must satisfy.

We say a function implementation is **correct** when:

> *For all inputs that satisfy the function specification's preconditions, the function implementation's return value satisfies the function specification's postconditions.*

Correctness (unary functions)

```python
def f(x: ____) -> ____:
    ...
```

- Let D be the set of possible inputs (e.g. `int`, `bool`, `dict`)
- Let $Pre : D \to \{True, False\}$ be the precondition(s) of `f`
- Let $Post : D \to \{True, False\}$ be the postcondition(s) of `f`

Note: Pre and $Post$ can be an AND of smaller predicates.

"`f`'s implementation is correct":

$$\forall x \in D, Pre(x) \Rightarrow Post(x)$$

Logical “filtering” revisited

$$\forall x \in D, Pre(x) \Rightarrow Post(x)$$

$Pre(x) \Rightarrow Post(x)$ is vacuously true when $Pre(x)$ is False.

If a function is called with inputs that do not satisfy the preconditions, the implementation **may or may not satisfy the postconditions**.

Type annotations as pre-/postconditions

```
def max_length(strings: set) -> int:
```

Parameter type annotations are a form of function precondition.

Return type annotation are a form of function postcondition.

Writing preconditions in docstrings

```
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the given
    strings.

    Preconditions:
      - strings != set()
    """
    return max({len(s) for s in strings})
```

Whenever possible, write preconditions as valid Python expressions that evaluate to a `bool`.

More specific collection type annotations

Type	Description
`set[T]`	A set whose elements all have type `T`. Example: `{'hi', 'bye'}` has type `set[str]`.
`list[T]`	A list whose elements all have type `T`. Example: `[1, 2, 3]` has type `list[int]`.
`dict[T1, T2]`	A dictionary whose keys all have type `T1` and whose associated values all have type `T2`. Example: `{'a': 1, 'b': 2, 'c': 3}` has type `dict[str, int]`.

Specific vs. general collection types

Use specific collection types (e.g. `set[str]`) when expecting a homogeneous collection.

Use general collection types (e.g. `set`) when:

- the collection could be heterogeneous, or
- the code does not depend on the type of the contained values (e.g., `len`)

Exercise 1: Reviewing preconditions and type annotations

Preconditions: function implementer vs. function caller

For the **implementer**:

A precondition is an assumption that makes the function easier to implement.

No need to worry about inputs that don't satisfy the precondition!

For the **caller**:

A precondition is a requirement that makes the function harder to call.

Need to make sure the arguments satisfy the precondition!

Sounds great, but...

What if the caller accidentally violates a precondition? (**Demo!**)

Warning: The Python interpreter does not check preconditions—even type annotations!

Checking preconditions with PythonTA

PythonTA and contract checking

PythonTA can automatically check function preconditions and postconditions!

1. Import the function `check_contracts` from the module `python_ta.contracts`.

   ```
   from python_ta.contracts import check_contracts
   ```

   This is an **import-from statement**, a variation of import statements that lets us use a specific variable/function from a module.

2. Add a line of code above the function definition we want to check:

```
@check_contracts
def calculate_pay(start: int, end: int, pay_rate: float)
    ...
```

`@check_contracts` is called a **decorator**, an optional part of a func
definition that adds additional behaviour to the function beyond wha
the function body.

Demo!

Demo!

```
>>> calculate_pay(1, 100, 15.0)
Traceback (most recent call last):
  ... [some lines omitted] ...
AssertionError: calculate_pay precondition "0 <= end <= 23
violated for arguments {start: 1, end: 100, pay_rate: 15.0
```

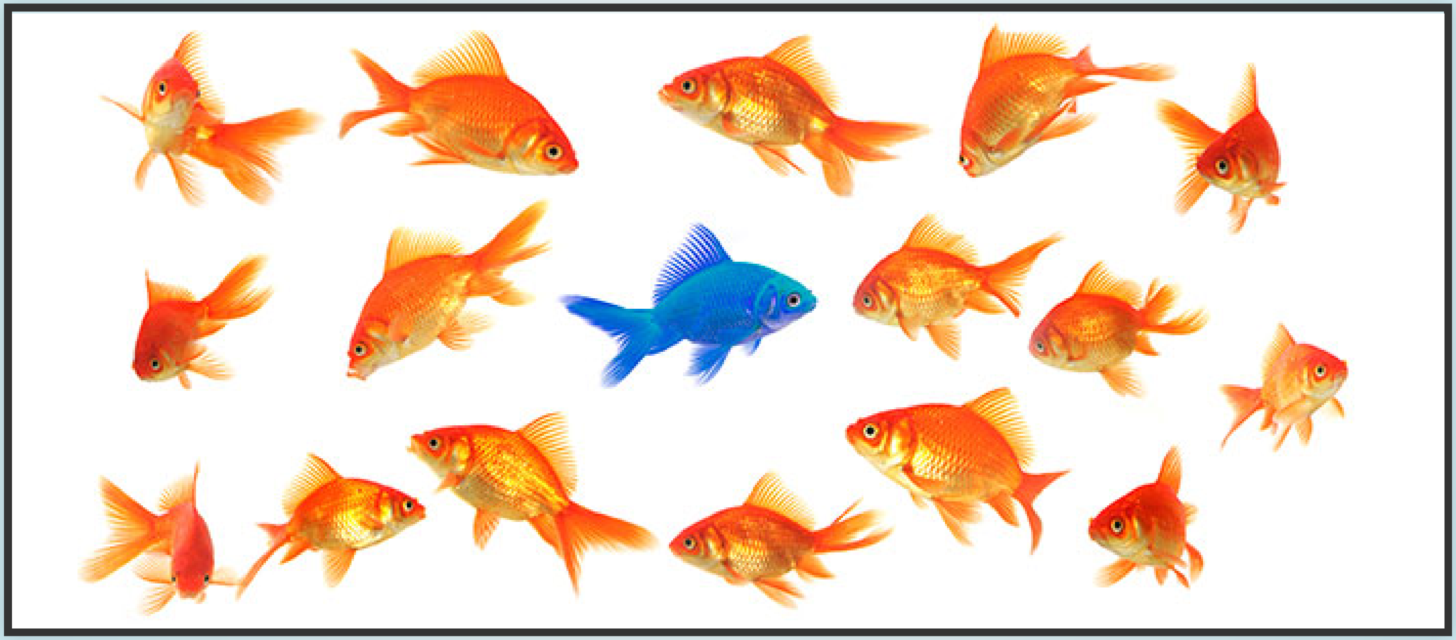
Property-based testing

Story so far

A **unit test** is a block of code that checks for the correct behaviour of a function for one specific input.

Both `doctest` and `pytest` use unit tests—though they are written in different ways!

Each unit test checks that a function’s implementation is correct for one input.



But recall our function correctness definition...

$$\forall x \in D, Pre(x) \Rightarrow Post(x)$$

Property-based test

A **property-based test** is a block of code that checks a property of a function on a large set of inputs.

Example

```
def is_even(value: int) -> bool:
    """Return whether value is divisible by 2."""
```

Unit tests:

```
def test_is_even_true() -> None:
    """Test is_even on an even number."""
    assert is_even(2)


def test_is_even_false() -> None:
    """Test is_even on an odd number."""
    assert not is_even(3)
```

Property-based tests

Property of `is_even`:

- `is_even` always returns `True` when given an `int` of the form `2 * x` (where `x` is an `int`)

$$\forall x \in \mathbb{Z},\ \textbf{is\_even}(2x)$$

Writing the test, part 1

```
def test_is_even_2x(x: int) -> None:
    """Test that is_even(2 * x) always returns True."""
    assert is_even(2 * x)
```

Problem: how do we tell `pytest` to “call” this test function on different values of `x`?

Using `hypothesis` to generate test inputs

`hypothesis` is a Python library for creating property-based tests. Its role is to take a test function and automatically generate inputs for that function.

Strategies (for generating data)

A **`hypothesis` strategy** is a data type that specifies a kind of value to generate for test input.

```
from hypothesis.strategies import integers
```

`integers` is a function that returns a strategy to generate “random” `int`s.

`given`

`from hypothesis import given`

`given` is another Python decorator. We use it to specify a strategy to generate inputs for a test function.

```
@given(x=integers())
def test_is_even_2x(x: int) -> None:
    """Test that is_even(2 * x) always returns True."""
    assert is_even(2 * x)
```

Demo!

Exercise 2: Property-based testing

Choosing properties

For unit tests: how do we know we have enough input-output pairs?

For property-based tests: how do we know we have enough properties?

Sometimes, it’s possible to prove that a collection of properties is enough!

Theorem (correctness for `is_even`). An implementation for `is_even` is correct if and only if it satisfies both of the following properties:

1. $\forall x \in \mathbb{Z},\ \texttt{is\_even}(2x) = True$
2. $\forall x \in \mathbb{Z},\ \texttt{is\_even}(2x + 1) = False$

See the end of Section 4.4 for another example of this type of theorem!

Proving Function Correctness

Suppose we want to write a function that calculates the sum of the first n positive integers.

```
def sum_to_n_v1(n: int):
    """Return the sum of the numbers from 1 to n, inclusiv
    """
    return sum([i for i in range(1, n + 1)])
```

```
def sum_to_n_v2(n: int):
    """Return the sum of the numbers from 1 to n, inclusiv
    """
    return n * (n + 1) // 2
```

Our two versions

Direct translation of the mathematical quantity we want to compute:

```
sum([i for i in range(1, n + 1)])
```

Something... else:

```
n * (n + 1) // 2
```

Theorem. For all $n \in \mathbb{Z}^+$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Mathematical proofs unlock new algorithms to solve problems. Often, these algorithms are faster than naive approaches.

More on this next class!

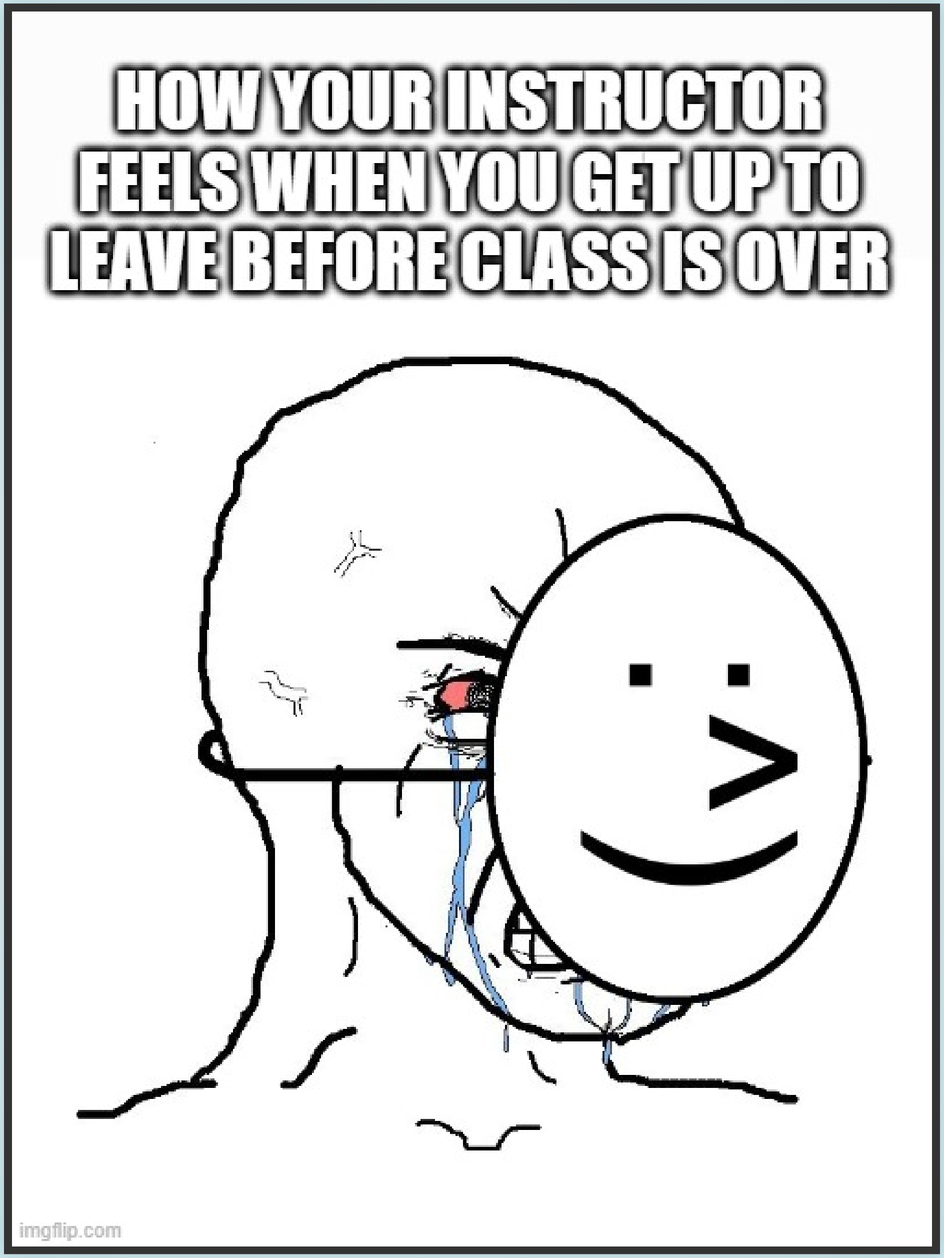
Summary

Today you learned to...

1. Define the terms precondition, postcondition, and function correctness.
2. Write detailed type annotations for collections in Python.
3. Document function preconditions as Python expressions in function docstrings.
4. Use PythonTA to check function preconditions and postconditions automatically.
5. Write property-based tests to check correctness of Python functions.
6. Differentiate between unit tests and property-based tests.

Homework

- Readings:
  - From prep: 4.1, 4.2
  - Today: 4.3, 4.4, 4.5
  - Next class: 4.6, 4.7
- Finish up Assignment 1
- Review for Term Test 1


HOW YOUR INSTRUCTOR FEELS WHEN YOU GET UP TO LEAVE BEFORE CLASS IS OVER
imgflip.com