CSC 148: Introduction to Computer Science

Week 2

Object-Oriented Programming (continued)

Representation invariants

Reminder: revisit the readings **before lecture** !

In class: apply content in exercises, discuss, ask questions

=> develop stronger command of the concepts!

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

Representation invariants

- Key point: How do we document the properties that must be true for every instance of a given class?

- Every instance attribute has a type annotation, which restricts the kind of value this attribute can have

- But we often want to restrict attributes values even further
  - Example: Tweets can have at most 280 characters

- What do we call these restrictions, and how do we communicate them?

Representation invariant

- A representation invariant is a property of the instance attributes that every instance of a class must satisfy

- How to express RIs?
  - (in words) This tweet's content is at most 280 characters.
  - (in code) `len(self.content) <= 280`

Let's get started!

- A representation invariant is a property of the instance attributes (including type annotations) that every instance of a class must satisfy

- Warm-up: complete the first page of today's worksheet

Today's take-aways: two questions about RIs

- 1. Why should we care about representation invariants?

- 2. How do we enforce representation invariants?

1. Representation invariants as assumptions

- A representation invariant is a property that every instance of a class must satisfy

- When given an instance of that class, we can assume that every representation invariant is satisfied

Representation invariants as assumptions

```
class Tweet:
    def like(self, n: int) -> None:
        self.likes += n
```

- `self` is an instance of `Tweet`, so we assume that all RIs are satisfied when this method is called
- The representation invariants of `Tweet` are **preconditions** of `self` for every `Tweet` method
  - e.g., like adding a precondition in docstring of like(): "self.likes is an int"
  - Preconditions of self for methods are generally **implied** from RIs (either explicit RIs in class docstring or implicit RIs from type contracts)
- What about `new_content` parameter for edit() method ?
- In general, the Zen of Python: "Explicit is better than implicit"

2. Enforcing representation invariants

- Every method must ensure that `self` satisfies all representation invariants after the method ends

- The representation invariants of a class are postconditions of `self` for every `Tweet` method

Strategy 1: Preconditions

- Require client code to call methods with “good” inputs, so that the methods won’t violate the representation invariants.

Strategy 2: Ignore “bad” inputs

- Accept a wide range of inputs, and if an input would cause a representation invariant to be violated, do nothing instead.

- Also known as failing silently.

Strategy 3: Fix “bad” inputs

- Accept a wide range of inputs, and if an input would cause a representation invariant to be violated, change it to a “reasonable” or default value before continuing with the rest of the function.

Discuss the pros and cons of each

- **Strategy 1**: use preconditions
- **Strategy 2**: ignore bad inputs
- **Strategy 3**: fix bad inputs

Direct attribute access

- Even if our methods are perfect, client code can access and mutate most instance attributes directly

- Documenting representation invariants is essential!

More design considerations …

- When adding some new feature in a class, consider what you already have and what you cannot implement without extra attributes and/or methods

- Remember: Redundant information is bad (memory space inefficiency, prone to bugs)

- Worksheet part 2

Learning Tips: What to do after lecture

- **Review**: re-read prep if needed, summarize, question, re-explain
- **Practice**: it's not enough to just read code we give you
- **Share**: meet with a friend or study group
- **Get help**: come to office hours!