**Useful Anthropic Interview Practical Advice**: <https://www.restack.io/p/anthropic-answer-interview-prep-cat-ai>

**Instructions**

Welcome to the practice task! This task is designed to be a playground that allows you to get acquainted with the testing environment and practice the functionalities of the platform before your real assessment. Your task is to implement a simple container of integer numbers. All operations that should be supported by this database are described below.

*Solving this task consists of several levels. Subsequent levels are opened when the current level is correctly solved. You always have access to the data for the current and all previous levels.*

**Environment details** (click here to hide)

To read the details about the language version, libraries used in the task, and the testing framework, used check out the **README** tabin the left panel of the page.

To customize the editor settings and see the editor hotkeys, check out the **SETTINGS** tabin the left panel of the page.

To reset your code, clickbutton in the top right corner of the page. Be careful, this will reset all your code changes!

Unit tests are already implemented and cannot be modified. You can find level 1 tests among the open files. Feel free to look at them and get familiar with the details. There is also an editable sandbox test file, which can be used for testing your code with custom test cases. These tests will be run together with other tests but will not be taken into account for scoring.

To run the tests, you can choose either **In Terminal** or **Structured** option from the dropdownand clickbutton located on the right part of the screen. The difference between these options is the format of the test results:

* If you'd like to receive a raw report of tests, choose the **In Terminal** option. The tests in this report will contain all debug output you've printed to the console.
* If you'd like to receive test results with more structure, choose the **Structured** option. Note that debug output is not supported in this option.

You can execute a single test case by running the following command in the terminal: **bash run\_single\_test.sh "<test\_case\_name>"**.

**Requirements**

Your task is to implement a simple container of integer numbers. Plan your design according to the level specifications below:

* **Level 1: Container should support adding and removing numbers.**
* Level 2: Container should support getting the median of the numbers stored in it.

To move to the next level, you need to pass all the tests at this level when submitting the solution.

**Level 1**

Implement two operations for adding and removing numbers from the container. Initially, the container is empty.

* **add(self, value: int) -> int** — should add the specified integer **value** to the container and return the number of integers in the container after the addition.
* **delete(self, value: int) -> bool** — should attempt to remove the specified integer **value** from the container. If the **value** is present in the container, remove it and return **True**, otherwise, return **False**.

**Examples**

The example below shows how these operations should work (the section is scrollable to the right):

|  |  |
| --- | --- |
| **Queries** | **Explanations** |
| **add(5)**  **add(10)**  **add(5)**  **delete(10)**  **delete(1)**  **add(1)** | **returns 1; container state: [5]**  **returns 2; container state: [5, 10]**  **returns 3; container state: [5, 10, 5]**  **returns True; container state: [5, 5]**  **returns False; container state: [5, 5]**  **returns 3; container state: [5, 5, 1]** |

* **[execution time limit] 3 seconds**
* **[memory limit] 1 GB**

import inspect, os, sys

current\_dir = os.path.dirname(os.path.abspath(inspect.getfile(inspect.currentframe())))

parent\_dir = os.path.dirname(current\_dir)

sys.path.insert(0, parent\_dir)

from timeout\_decorator import timeout

import unittest

from integer\_container\_impl import IntegerContainerImpl

class SandboxTests(unittest.TestCase):

"""

The test class below can be considered as a playground - feel free to modify it as you need, e.g.:

- add your own custom tests

- delete existing tests

- modify test contents or expected output

The results of tests from this file will always be at the beginning of the report generated by clicking the "Run" button.

The results of these tests do not affect the final score (unless the project fails to build).

"""

failureException = Exception

@classmethod

def setUp(cls):

cls.container = IntegerContainerImpl()

@timeout(0.4)

def test\_sample(self):

self.assertEqual(self.container.add(5), 1)

self.assertEqual(self.container.add(10), 2)

self.assertEqual(self.container.add(5), 3)

self.assertTrue(self.container.delete(10))

self.assertFalse(self.container.delete(1))

self.assertEqual(self.container.add(1), 3)

What does this do or class vs. instance-level methods?

@classmethod indicates that setUp is bound to the class (not an instance).

Because it’s a class method, the first parameter is conventionally named cls instead of self.

When setUp(cls) runs, it creates one container on the class itself: cls.container. All test methods within this class can then access the same container via cls.container (or self.container, which resolves to the same class attribute).

In contrast, if you used an instance method:

python

Copy

def setUp(self):

self.container = IntegerContainerImpl()

Each test method would get its own separate container, because self refers to a new instance of the test class for every test method.

How is cls different from self?

self refers to the current instance of a class. In the context of unittest.TestCase, each test method runs in its own new instance, so self attributes are unique per test.

cls refers to the class itself, so attributes set on cls become class attributes—they are shared by all instances (i.e., all test methods) of that test class.

In short:

@classmethod def setUp(cls): ...

A one-time setup for the entire test class, storing data in a class attribute.

def setUp(self): ...

An instance-level setup that runs before each test method, storing data in an instance attribute.

**This is how the test method would look if we use an instance-level variable:**  
import unittest

from timeout\_decorator import timeout

class SandboxTests(unittest.TestCase):

failureException = Exception

def setUp(self):

self.container = IntegerContainerImpl()

@timeout(0.4)

def test\_sample(self):

self.assertEqual(self.container.add(5), 1)

self.assertEqual(self.container.add(10), 2)

self.assertEqual(self.container.add(5), 3)

self.assertTrue(self.container.delete(10))

self.assertFalse(self.container.delete(1))

self.assertEqual(self.container.add(1), 3)

**Key Differences**

* **Decorator**: You remove @classmethod, so now setUp is an instance method rather than a class method.
* **Parameter**: Instead of def setUp(cls):, it becomes def setUp(self):.
* **Attribute Storage**: Instead of cls.container, you use self.container.

With this change, each test method gets its own fresh self.container instance.

**But there is no difference with the original code? Is the logic of the code the same**?

Reasoned for 22 seconds

If you only have **one** test method (test\_sample), then in practice there’s no visible difference: both versions create a container before running the single test, and the test behaves the same.

However, if you were to add **multiple** test methods in the same class, the difference becomes clear:

* **Class-level setup** (@classmethod def setUp(cls): …)
  + The container is created **once** at the class level.
  + All test methods in that class share the **same** container.
  + One test’s changes might affect subsequent tests.
* **Instance-level setup** (def setUp(self): …)
  + The container is created **before each test**.
  + Each test gets a **fresh** container, so tests do not affect each other.

So logically, for a **single** test method, the code’s behavior is effectively the same. For **multiple** tests, the instance-level setup ensures isolation between tests, while a class-level setup causes all tests to use the same container.

Solution class:  
  
class IntegerContainerImpl:

def \_\_init\_\_(self):

# We use a list to store the integers.

self.data = []

def add(self, value: int) -> int:

# Append the new value.

self.data.append(value)

# Return the current total number of elements.

return len(self.data)

def delete(self, value: int) -> bool:

# Check if the value exists.

if value in self.data:

# Remove one occurrence of the value.

self.data.remove(value)

return True

return False

Continued