CSC110 Lecture 28: Inheritance

Hisbaan Noorani

November 25, 2020

Contents

1 Exercise 1: Inheritance

1. Please use the following questions to review the terminology we have covered so far this lecture.

   (a) What is an **abstract method**?

   A method that is declaired but does not have an implementation. The body of this method will raise a `NotImplementedError`.

   (b) What is an **abstract class**?

   A class that contains one or more abstract methods.

   (c) Consider the following Python class. Is it abstract or concrete?

```
class MyClass:

    def do_something(x: int) -> int:
        return x + 5

    def do_something_else(y: int) -> int:
        raise NotImplementedError
```

   Abstract. `MyClass.do_something` is an abstract method.

2. Consider the Stack inheritance hierarchy introduced in lecture, where the abstract class `Stack` is the parent class of both `Stack1` and `Stack2`. For each of the following code snippets in the Python console, write the output or describe the error that would occur, and explain.

   (a)

```
>>> s = Stack2()
>>> isinstance(s, Stack1)
False
```

   `Stack2` and `Stack1` are both children of `Stack` but that does not mean that if an object is an instance of `Stack2` then it is also an instance of `Stack1`. They are siblings but not twins.

(b)
```
>>> s = Stack1()
>>> Stack.push(s, 'book')
>>> Stack.pop(s)
NotImplementedError
```

Since this method is not implemented in the `Stack` class, python will raise a `NotImplementedError`. We should be calling from `Stack1`.

(c)
```
>>> s = Stack()
>>> s.push('paper')
NotImplementedError
```

The `Stack` class itself does not have an implementation of its methods. It relies on its child classes so one of those should be initialized instead.

3. We have said that inheritance serves as another form of *contract*:

   - The implementor of the subclass must implement the methods from the abstract superclass.
   - Any user of the subclass may assume that they can call the superclass methods on instances of the subclass.

   What happens if we violate this contract? Once again, consider the classes `Stack` and `Stack1`. Expect this time, the method `Stack1.is_empty` is missing:

```
class Stack1(Stack):
    """..."""
    # Private Instance Attributes
    #   _items: The elements in the stack
    _items: List[Any]

    def __init__(self) -> None:
        """Initialize a new empty stack."""
        self._items = []

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """
        list.append(self._items, item)

    def pop(self) -> Any:
        """Remove and return the element at the top of this stack.

        Preconditions:
            - not self.is_empty()
        """
        return list.pop(self._items)
```

Try executing the following lines of code in the Python console—what happens?

```
>>> s = Stack1()
>>> s.push('pancake')
>>> s.is_empty()
```

This will raise a `NotImplementedError`. This class violates the contract. It does not implement the `Stack.is_empty` method so it inherits the one that is not implemented.

2 Exercise 2: Polymorphism

```
def weird(stacks: List[Stack]) -> None:
    for stack in stacks:
        if stack.is_empty():
            stack.push('pancake')
        else:
            stack.pop()
```

1. Suppose we execute the following code in the Python console:

```
>>> list_of_stacks = [Stack1(), Stack2(), Stack1(), Stack2()]
>>> list_of_stacks[0].push('chocolate')
>>> list_of_stacks[2].push('chocolate')
```

Now suppose we call `weird(list_of_stacks)`. Given the list `list_of_stacks`, write the specific `push` or `pop` method that would be called at each loop iteration. The first is done for you.

`weird` iteration	`push=/=pop` version
0	`Stack1.pop`
1	`Stack2.push`
2	`Stack1.pop`
3	`Stack2.push`

2. Write a code snippet in the Python console that results in a variable `list_of_stacks2` that, if passed to `weird`, would result in the following sequence of `push=/=pop` method calls: `Stack1.push`, `Stack2.push`, `Stack1.pop`, `Stack2.pop`.

```
>>> list_of_stacks2 = [Stack1(), Stack2(), Stack1(), Stack2()]
>>> list_of_stacks2[2].push('chocolate')
>>> list_of_stacks2[3].push('chocolate')
```

3. Create a list `list_of_stacks3` that, if passed to `weird`, would raise a `NotImplementedError` on the second loop iteration.

```
>>> list_of_stacks3 = [Stack1(), Stack]
```

3 Exercise 3: The `object` superclass and overriding methods

1. Does our `Stack` abstract class have a parent class? If so, what is it? If not, why not?

   Yes, it is the `object` class.

2. Suppose we have a variable `my_stack = Stack1()`. What information does the string representation `str(my_stack)` display?

   It will represent the type of stack, as well as the items in the stack.

   In this case, that will be `'Stack1: empty'`.

3. In the space below, override the `__str__` method for the `Stack1` class, so that the string representation matches the format shown in the docstring.

   Note: You should call `str` on each item stored in the stack.

```
class Stack1:
    _items: list

    # ... other code omitted

    def __str__(self) -> str:
        """Return a string representation of this stack.

        >>> s = Stack1()
        >>> str(s)
        'Stack1: empty'
        >>> s.push(10)
        >>> s.push(20)
        >>> s.push(30)
        >>> str(s)
        'Stack1: 30 (top), 20, 10'

        Notes:
            - because this is a method, you may access the _items attribute
            - call str on each element of the stack to get string representations
              of the items
            - review the str.join method
            - you can reverse the items in a list by calling reversed on it
              (returns a new iterable) or the list.reverse method (mutates the list)
        """
        str_so_far = 'Stack1: '
        if self.is_empty():
            str_so_far += 'empty'
        else:
            strings = [str(item) for item in self._items]
            list.reverse(strings)
            strings[0] = strings[0] + ' (top)'
            str_so_far += ', '.join(strings)

        return str_so_far
```