CSC110 Lecture 28: Inheritance

Print this handout

Exercise 1: Inheritance

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

   a. What is an **abstract method**?

   A method that raises the NotImplementedError.
   (We talk about it 'in the abstract' but don't use it.)

   b. What is an **abstract class**?

   A class that defines (or inherits, without overriding) at least one abstract method.

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

```python
class MyClass:

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

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

   Abstract, since it has a method (do_something_else) that is abstract.

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

since Stack2 is not a child of Stack1.

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

The NotImplementedError would be raised since the abstract method Stack.push would be called.

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

As in (b) since Stack is an abstract class and Stack.push is not implemented.

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`, exce
this time, the method `Stack1.is_empty` is missing:

```

```

```python
class Stack1(Stack):
    """..."""
    # Private Instance Attributes
    #   _items: The elements in the stack
    _items: list

    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.
        """
        self._items.append(item)

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

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

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

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

The method Stack.is_empty is inherited but it is abstract, so a NotImplementedError is raised.

Exercise 2: Polymorphism

Consider the function `weird` below:

```
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 specif `push` or `pop` method that would be called at each loop iteration. The first is done for you.

weird loop 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('maple')
list_of_stacks2[3].push('strawberry')
```

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

list_of_Stacks3 = [Stack1(), Stack()]

Additional Exercise: 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?

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

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.

```python
class Stack1(Stack):
    _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)
        """
```