10.5 Stacks

Over the next few sections, we'll learn about three new abstract data types: Stack, Queue, and Priority Queue. All three of these ADTs store a collection of items, and support operations to add an item and remove an item. However, unlike a Set or List, in which users may specify which item to remove (by value or by index, respectively), these three ADTs remove and return their items in a fixed order—client code is allowed no choice. This might seem restrictive and simplistic, but you'll soon learn how the power of these ADTs lies in their simplicity. Once you learn about them, you'll start seeing them everywhere, and be able to effectively communicate about these ADTs to any other computer scientist.

The Stack ADT

Photo credit: https://www.pexels.com/

The **Stack** ADT is very simple. A stack contains zero or more items. When you add an item, it goes "on the top" of the stack (we call this "pushing" onto the stack) and when you remove an item, it is removed from the top also (we call this "popping" from the stack).[1] The net effect is that the first item added to the stack is the last item removed. We call this *Last-In-First-Out (LIFO)* behaviour. To summarize:

[1] The name "stack" is a deliberate metaphor for a stack of books on a table.

- **Stack**
  - Data: a collection of items
  - Operations: determine whether the stack is empty, add an item (*push*), remove the most recently-added item (*pop*)

In code:

```
from typing import Any


class Stack:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in last-in, first-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    Sample usage:

    >>> s = Stack()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """

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

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """

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

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

        Preconditions:
            - not self.is_empty()
        """
```

At this point, you may be wondering how we fill in the method bodies, picturing perhaps a `list` instance attribute to store the items in the stack. But remember, thinking about implementation is irrelevant when you are using an ADT. At this point, you should picture a pile of objects stacked on top of each other—this is enough to understand each of the doctest examples in the above code. Abstraction allows us to separate our understanding of what the Stack ADT is from how it is implemented.

Applications of stacks

Because they have so few methods, it may seem like stacks are not that powerful. But in fact, stacks are useful for many things. For instance, they can be used to check for balanced parentheses in a mathematical expression. And consider the execution of a Python program. We have talked about frames that store the names available at a given moment in its execution. What happens when `f` calls `g`, which calls `h`? When `h` is over, we go back to `g` and when `g` is over we go back to `f`. To make this happen, our frames go on a stack! Hence the names *call stack* and *stack frame* from our memory model.

As a more "real world" example, consider the undo feature in many different applications. When we perform an action by mistake and want to undo it, we want to undo *the most recent* action, and so the Stack ADT is the perfect abstract data type for keeping track of the history of our actions so that we can undo them. A similar application lies in how web browsers store page visits so that we can go back to the most recently-visited page.

Implementing the Stack ADT using lists

Next, we'll now implement the Stack ADT using a built-in Python data structure: the `list`. We've chosen to use the *end* of the list to represent the top of the stack.

```
from typing import Any


class Stack1:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    Instance Attributes:
        - items: The items stored in the stack. The end of the list represents
          the top of the stack.

    >>> s = Stack1()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    items: list

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

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """
        return 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()
```

Attributes and the class interface

Our current `Stack1` class is correct, but has one subtle difference with the Stack ADT it is supposed to implement. While a user can create a new `Stack1` object and call its methods `push` and `pop` to interact with it, they can also do one more thing: access the `items` instance attribute. This means that any user of a `Stack1` object can access any item in the stack at any time, or even mutate `items` to modify the contents of the stack in unexpected ways.

You might wonder why this is an issue—if a user wants to change the `items` attribute, let them! And indeed this is a common and valid approach in programming, particularly in favour with many Python developers. However, it is not the only approach. Another school of thought is that a data type's interface should communicate not just how to use it, but also how *not* to use it. For our current `Stack1` implementation, the instance attribute `items` is part of the class' interface, and so all users can reasonably expect to use it.

To make an instance attribute that *isn't* part of a class' interface, we prefix its name with an underscore `_`. We refer to attributes whose names begin with an underscore as **private instance attributes**, and those without the underscore (all the attributes we've seen so far) as **public instance attributes**. These names suggest how they're interpreted when it comes to a class interface: all public instance attributes are part of the interface, and all private ones aren't.[2]

[2] When designing classes professionals often make distinctions between three levels of attribute access: *public*, *private*, and *protected*. To keep things simple we'll only be using the first two terms in this course, even though our notion of "private" overlaps with how "protected" is commonly used.

Here's how we could modify our `Stack1` implementation to make `items` a private attribute instead.

```
class Stack1:
    """A last-in-first-out (LIFO) stack of items.

    Stores data in first-in, last-out order. When removing an item from the
    stack, the most recently-added item is the one that is removed.

    >>> s = Stack1()
    >>> s.is_empty()
    True
    >>> s.push('hello')
    >>> s.is_empty()
    False
    >>> s.push('goodbye')
    >>> s.pop()
    'goodbye'
    """
    # Private Instance Attributes:
    #   - _items: The items stored in the stack. The end of the list represents
    #     the top of the stack.
    _items: list

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

    def is_empty(self) -> bool:
        """Return whether this stack contains no items.
        """
        return 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()
```

Other than renaming the attribute from `items` to `_items`, the only change is in how we document this attribute. We've kept the same format, but now moved the description from the class docstring to comments in the class body. By doing so, there is now no mention of this attribute when we call `help` on our class:

```
>>> help(Stack1)
class Stack1(builtins.object)
 |  Stack1() -> None
 |
 |  A last-in-first-out (LIFO) stack of items.
 |
 |  Stores data in a last-in, first-out order. When removing an item from the
 |  stack, the most recently-added item is the one that is removed.
 |
 |  >>> s = Stack1()
 |  >>> s.is_empty()
 |  True
 |  >>> s.push('hello')
 |  >>> s.is_empty()
 |  False
 |  >>> s.push('goodbye')
 |  >>> s.pop()
 |  'goodbye'
 |
 |  [The rest is omitted]
 |
```

Warning: private attributes can be accessed!

One of the distinctive features of Python that separates it from many other programming languages is that private instance attributes can still be accessed from outside the class.

```
>>> s = Stack1()
>>> s.push(10)
>>> s.push(20)
>>> s._items
[10, 20]
```

This is a design choice made by the creators of the Python programming language to prefer flexibility over restriction when it comes to accessing attributes. But does this mean private attributes are meaningless? **No!** By making an instance attribute private, we are communicating that client code should *not* access this attribute: it is not an expected way of interacting with this class. As a result, we reduce the cognitive load on the client (one less attribute to think about when using the class), and also give flexibility to the designer of the class to change or even remove a private attribute if they want to update their implementation of the class, without affecting the class' public interface.

Comparing stack implementations with running-time analysis

We implemented `Stack1` using the back of the `_items` list to represent the top of the stack. You might wonder why we didn't use the front of `_items` instead. Indeed, the implemention wouldn't have to change much:

```
class Stack2:
    # Duplicated code from Stack1 omitted. Only push and pop are different.

    def push(self, item: Any) -> None:
        """Add a new element to the top of this stack.
        """
        self._items.insert(0, 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(0)
```

The key difference between `Stack1` and `Stack2` is not their code complexity but their *efficiency*. In Chapter 9, we learned that Python uses an array-based implementation for lists. Because of this, the `list.append` operation for an array-based list is $\Theta(1)$, therefore `Stack1.push` also has a $\Theta(1)$ running time. In contrast, `list.insert` has complexity $\Theta(n - i)$, where i is the index argument passed to `list.insert`. In `Stack2.push`, $i = 0$ and so the method has complexity $\Theta(n)$. So the `push` operation for stacks is more efficient when we treat the end of an array-based list as the top of the stack.

Similarly, removing the last element of an array-based list using `list.pop` is also $\Theta(1)$, and so the running time of `Stack1.pop` is $\Theta(1)$. However, `Stack2.pop` uses passes an index of 0 to `list.pop`, which causes the method to have a $\Theta(n)$ running time.

The decision of which implementation has superior efficiency is clear: `Stack1` will always be more efficient than `Stack2`. Having such a clear-cut winner is actually quite rare. There are almost always trade-offs associated with choosing one implementation over another. We will see one such trade-off when we introduce our next ADT: queues.