10.9 Defining a Shared Public Interface with Inheritance

In this chapter, we learned that an abstract data type can have multiple implementations, and saw this first-hand with a variety of ADTs. For example, in 10.5 Stacks we saw that the Stack ADT can be implemented using a Python list in two different ways, storing the top of the stack at the end of the list (`Stack1`) or the front of the list (`Stack2`). Though these two classes had different implementations, they shared the same *public interface* of the Stack ADT.

One limitation of the code we wrote for these two classes is that the only way to tell that `Stack1` and `Stack2` had the same interface was from their method names and docstrings. In this section, we'll see how to create a special kind of Python class that is used to define a public interface that can be implemented by other classes, using a Python language feature known as *inheritance*.

The Stack abstract class

Let us begin by defining a `Stack` class that consists only of the *public interface* of the Stack ADT.

```python
from typing import Any


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

    This is an abstract class. Only subclasses should be instantiated.
    """

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

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

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

        Raise an EmptyStackError if this stack is empty.
        """
        raise NotImplementedError


class EmptyStackError(Exception):
    """Exception raised when calling pop on an empty stack."""
```

In Python, we mark a method as unimplemented by having its body raise a special exception, `NotImplementedError`. We say that a method is **abstract** when it is not implemented and raises this error; we say that a *class* is **abstract** when at least one of its methods is abstract (i.e., not implemented). A **concrete class** is a class that is not abstract; so far in this course, we've been dealing with concrete classes, and called them concrete data types.[1]

[1] The terminology here is a bit confusing because of the multiple uses of certain terms. A concrete Python class is the same as a concrete data type. However, an abstract Python class is *not* the same thing as an abstract data type; the former has a technical meaning specific to the Python programming language, while the latter is the name given to an abstract description of a data type that is programming language-independent.

Now, you might wonder what the purpose of an abstract class is. Indeed, a programmer who creates a `Stack` object will quickly find it is useless, because calling the Stack ADT operations cause errors:

```python
>>> s = Stack()
>>> s.push(30)
Traceback...
NotImplementedError
>>> s.pop()
Traceback...
NotImplementedError
```

If we can't use the `Stack` object for any of the Stack ADT operations, what was the point in defining this class? The answer is very much based on abstraction, hence the name abstract class. The `Stack` class we have defined is a direct translation of the Stack ADT: an **interface** that describes the methods that a concrete class that wants to implement the Stack ADT *must* define. Python gives us a way to describe the relationship between an abstract class and a concrete class that implements its methods directly in the code.

Inheriting the Stack abstract class

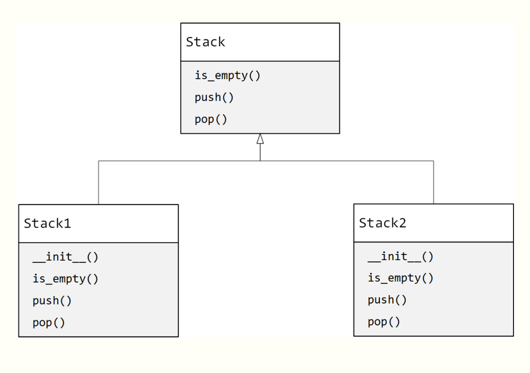
Earlier in this chapter, we defined two new types: `Stack1` and `Stack2`. However, despite the two types sharing the same method names, the code did not indicate that the types were related in any way. Now that we have the abstract class `Stack`, we can indicate this relationship in the code through **inheritance**:

```python
class Stack1(Stack):
    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """..."""
        return self._items == []

    def push(self, item: Any) -> None:
        """..."""
        self._items.append(item)

    def pop(self) -> Any:
        """..."""
        return self._items.pop()


class Stack2(Stack):
    def __init__(self) -> None:
        """Initialize a new empty stack.
        """
        self._items = []

    def is_empty(self) -> bool:
        """..."""
        return self._items == []

    def push(self, item: Any) -> None:
        """..."""
        self._items.insert(0, item)

    def pop(self) -> Any:
        """..."""
        return self._items.pop(0)
```

In the class header `class Stack1(Stack)` and `class Stack2(Stack)`, the syntax `(Stack)` indicates that `Stack1` and `Stack2` inherit from `Stack`. There are specific words we use to talk about these relationships:

- `Stack`: **superclass** and **parent class** are synonyms.
- `Stack1`, `Stack2`: **subclass** and **child class** are synonyms.

For example, we can say that "`Stack` is the parent class of `Stack1`" or "`Stack2` is a subclass of `Stack`".



When one class in Python inherits from another, there are two important consequences. First, the Python interpreter treats every instance of the subclass as an instance of the superclass as well.

```python
>>> stack1 = Stack1()
>>> isinstance(stack1, Stack1)
True
>>> isinstance(stack1, Stack)
True
>>> isinstance(stack1, Stack2)
False
```

Second, when the superclass is abstract, the subclass must implement all abstract methods from the superclass, without changing the public interface of those methods. Just like preconditions and representation invariants, inheritance serves as another form of *contract*:

- The creator 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.

So for example, if we say that `Stack1` is a subclass of `Stack`, then any user of `Stack1` can expect to be able to call `push`, `pop`, and `is_empty` on `Stack1` instances. And of course the same applies to `Stack2` as well.

It is this expectation that allows us to use inheritance in Python to express a *shared public interface* between multiples classes. In our example, because `Stack1` and `Stack2` are both subclasses of `Stack`, we expect them implement all the stack methods. They might also implement additional methods that are unique to each subclass (*not* shared), but this is not required.

Writing polymorphic code using inheritance

Suppose we are writing code that operates on a stack, like in the following function:

```python
def push_and_pop(stack: ..., item: Any) -> None:
    """Push and pop the given item onto the given stack.""
    stack.push(item)
    stack.pop()
```

What type annotation would be appropriate for `stack`? If we use a concrete stack implementation like `Stack1`, this would rule out other stack implementations for this function. Instead, we use the abstract class `Stack` as the type annotation, to indicate that our function `push_and_pop` can be called with *any* instance of any `Stack` subclass.

```python
def push_and_pop(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack.""
    stack.push(item)
    stack.pop()
```

Remember that `Stack` defines a public interface that is shared between all of its subclasses: the body of `push_and_pop` only needs to call methods from that interface (`pop` and `push`), and doesn't worry about how those methods are implemented. This allows us to pass to the `push_and_pop` function a `Stack1` or `Stack2` object, which both inherit from `Stack`.

```python
>>> stack1 = Stack1()
>>> push_and_pop(stack1)  # This works!
>>> stack2 = Stack2()
>>> push_and_pop(stack2)  # This also works!
```

You might notice that there are actually three versions of `push` in our code: `Stack.push`, `Stack1.push`, and `Stack2.push`. So which method does the Python interpreter choose when the `push_and_pop` function is called? This is how it works for `stack.push(item)` (`stack.pop()` is handled similarly):

1. When the Python interpreter evaluates `stack.push(item)`, it first computes `type(stack)`. The result will depend on the argument we passed in—in our above example, `type(stack1)` is `Stack1`, and `type(stack2)` is `Stack2`.
2. The Python interpreter then looks in that class for a `push` method and calls it, passing in `stack` for the `self` argument.[2]

[2] There are instances with inheritance where a subclass might not implement a particular method from the superclass. We'll look at some examples of this in the next section.

We say that the Python interpreter *dynamically looks up (or resolves) the `stack.push`/`stack.pop` method*, because the actual method called by `stack.push`/`stack.pop` changes depending on the argument passed to `push_and_pop`.

We say that the `push_and_pop` function is **polymorphic**, meaning it can take as inputs values of different concrete data types and select a specific method based on the type of input. This support for polymorphism is also why the "object dot notation" style of method call is preferred to the "class dot notation" style we've been using up to this point. Consider the following two alternate implementations of `push_and_pop`:

```python
def push_and_pop_alt1(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack.""
    Stack.push(stack, item)
    Stack.pop(stack)


def push_and_pop_alt2(stack: Stack, item: Any) -> None:
    """Push and pop the given item onto the stack stack.""
    Stack1.push(stack, item)
    Stack1.pop(stack)
```

The first version (`alt1`) explicitly calls the `Stack.push` and `Stack.pop` methods, both of which are unimplemented and would raise a `NotImplementedError`. The second version (`alt2`) calls concrete methods `Stack1.push` and `Stack1.pop`, which assumes a specific stack implementation (`Stack1`), and so `push_and_pop` would only be guaranteed to work on `Stack1` instances, but not any other `Stack` subclass. This makes `push_and_pop` no longer polymorphic: the correct type annotation for `s` would be `Stack1`, not `Stack`.

Application: running timing experiments on stack implementation

Because both `Stack1` and `Stack2` are different implementations of the same interface, we can use polymorphism to help us measure the performance of each. Below, we time the `push_and_pop` function, first with a `Stack1` object and second with a `Stack2` object.

```python
if __name__ == '__main__':
    # Import the main timing function.
    from timeit import timeit

    # The stack sizes we want to try.
    STACK_SIZES = [1000, 10000, 100000, 1000000, 10000000]
    for stack_size in STACK_SIZES:
        stack1 = Stack1()
        stack2 = Stack2()

        # Bypass the Stack interface to create a stack of size <stack_size>.
        # This speeds up the experiment, but we know this violates
        # encapsulation!
        stack1._items = list(range(0, stack_size))
        stack2._items = list(range(0, stack_size))

        # Call push_and_pop(stack1) 1000 times, and store the time taken.
        t1 = timeit('push_and_pop(stack1, 10)', number=1000, globals=globals())
        t2 = timeit('push_and_pop(stack2, 10)', number=1000, globals=globals())

        print(f'Stack size {stack_size:>8}; Stack1 time {t1}; Stack2 time {t2}')
```

If we have several implementations of an ADT, each inheriting from the same base class, then we can quickly run experiments on all of them but only need to remember a single interface. This creates a rule of thumb: when indicating the type of an object (e.g., through a type contract), choose the most generic type possible. Following this rule of thumb means that the client code is not constrained to one particular implementation (such as `Stack1`) and can readily change the underlying object so long as the new object type shares the same public interface.

Many software applications follow the same principle. For example, you may have used software with "plugins":[3] each plugin implements a shared public interface, allowing the software to use it without knowing any of the details. For example, Adobe develops the powerful Photoshop application for image editing. David comes along and discovers a feature he really wants is missing. Rather than asking Adobe to implement the new feature, he can implement it himself as a plugin. Thus, Adobe has allowed independent developers to *extend the functionality* of their software after it has been released and without any employees of their own. Behold, the power of abstraction!

[3] Like PyCharm!