Stacks

19.1. Abstract data types

The data types you have seen so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the Card class represents a card using two integers. As we discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do), but it does not not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

1. It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
2. Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.
3. Well-known ADTs, such as the Stack ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
4. The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called the **provider** code.

19.2. The Stack ADT

In this chapter, we will look at one common ADT, the **stack**. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include dictionaries and lists.

An ADT is defined by the operations that can be performed on it, which is called an **interface**. The interface for a stack consists of these operations:

__init__

Initialize a new empty stack.

push

Add a new item to the stack.

pop

Remove and return an item from the stack. The item that is returned is always the last one that was added.

is_empty

Check whether the stack is empty.

A stack is sometimes called a last in, first out or LIFO data structure, because the last item added is the first to be removed.

19.3. Implementing stacks with Python lists

The list operations that Python provides are similar to the operations that define a stack. The interface isn't exactly what it is supposed to be, but we can write code to translate from the Stack ADT to the built-in operations.

This code is called an **implementation** of the Stack ADT. In general, an implementation is a set of methods that satisfy the syntactic and semantic requirements of an interface.

Here is an implementation of the Stack ADT that uses a Python list:

```
class Stack :
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def is_empty(self):
        return (self.items == [])
```

A Stack object contains an attribute named items that is a list of items in the stack. The initialization method sets items to the empty list.

To push a new item onto the stack, push appends it onto items. To pop an item off the stack, pop uses the homonymous (*same-named*) list method to remove and return the last item on the list.

Finally, to check if the stack is empty, is_empty compares items to the empty list.

An implementation like this, in which the methods consist of simple invocations of existing methods, is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

19.4. Pushing and popping

A stack is a **generic data structure**, which means that we can add any type of item to it. The following example pushes two integers and a string onto the stack:

```
>>> s = Stack()
>>> s.push(54)
>>> s.push(45)
>>> s.push("+")
```

We can use is_empty and pop to remove and print all of the items on the stack:

```
while not s.is_empty():
    print s.pop(),
```

The output is + 45 54. In other words, we just used a stack to print the items backward! Granted, it's not the standard format for printing a list, but by using a stack, it was remarkably easy to do.

You should compare this bit of code to the implementation of print_backward in the last chapter. There is a natural parallel between the recursive version ofprint_backward and the stack algorithm here. The difference is that print_backward uses the runtime stack to keep track of the nodes while it traverses the list, and then prints them on the way back

from the recursion. The stack algorithm does the same thing, except that is use a Stack object instead of the runtime stack.

19.5. Using a stack to evaluate postfix

In most programming languages, mathematical expressions are written with the operator between the two operands, as in 1 + 2. This format is called **infix**. An alternative used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in 1 2 +.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack:

1. Starting at the beginning of the expression, get one term (operator or operand) at a time.
    - If the term is an operand, push it on the stack.
    - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
2. When you get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

19.6. Parsing

To implement the previous algorithm, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results—the individual chunks of the string – are called **tokens**. You might remember these words from Chapter 1.

Python provides a split method in both the string and re (regular expression) modules. The function string.split splits a string into a list using a single character as a**delimiter**. For example:

```
>>> import string
>>> string.split("Now is the time"," ")
['Now', 'is', 'the', 'time']
```

In this case, the delimiter is the space character, so the string is split at each space.

The function re.split is more powerful, allowing us to provide a regular expression instead of a delimiter. A regular expression is a way of specifying a set of strings. For example, [A-z] is the set of all letters and [0-9] is the set of all numbers. The ^ operator negates a set, so [^0-9] is the set of everything that is not a number, which is exactly the set we want to use to split up postfix expressions:

```
>>> import re
>>> re.split("([^0-9])", "123+456*/")
['123', '+', '456', '*', '', '/', '']
```

Notice that the order of the arguments is different from string.split; the delimiter comes before the string.

The resulting list includes the operands 123 and 456 and the operators * and /. It also includes two empty strings that are inserted after the operands.

19.7. Evaluating postfix

To evaluate a postfix expression, we will use the parser from the previous section and the algorithm from the section before that. To keep things simple, we'll start with an evaluator that only implements the operators + and *:

```
def eval_postfix(expr):
    import re
    token_list = re.split("([^0-9])", expr)
    stack = Stack()
    for token in token_list:
        if token == '' or token == ' ':
            continue
        if token == '+':
            sum = stack.pop() + stack.pop()
            stack.push(sum)
        elif token == '*':
            product = stack.pop() * stack.pop()
            stack.push(product)
        else:
```

```
            stack.push(int(token))
    return stack.pop()
```

The first condition takes care of spaces and empty strings. The next two conditions handle operators. We assume, for now, that anything else must be an operand. Of course, it would be better to check for erroneous input and report an error message, but we'll get to that later.

Let's test it by evaluating the postfix form of (56+47)*2:

```
>>> print eval_postfix ("56 47 + 2 \*")
206
```

That's close enough.

19.8. Clients and providers

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct – in accord with the specification of the ADT – and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Python's built-in types, you have the luxury of thinking exclusively as a client.

Of course, when you implement an ADT, you also have to write client code to test it. In that case, you play both roles, which can be confusing. You should make some effort to keep track of which role you are playing at any moment.

19.9. Glossary

abstract data type (ADT)
A data type (usually a collection of objects) that is defined by a set of operations but that can be implemented in a variety of ways.

interface
The set of operations that define an ADT.

implementation
Code that satisfies the syntactic and semantic requirements of an interface.

client

A program (or the person who wrote it) that uses an ADT.

provider

The code (or the person who wrote it) that implements an ADT.

veneer

A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

generic data structure

A kind of data structure that can contain data of any type.

infix

A way of writing mathematical expressions with the operators between the operands.

postfix

A way of writing mathematical expressions with the operators after the operands.

parse

To read a string of characters or tokens and analyze its grammatical structure.

token

A set of characters that are treated as a unit for purposes of parsing, such as the words in a natural language.

delimiter

A character that is used to separate tokens, such as punctuation in a natural language.

19.10. Exercises

Apply the postfix algorithm to the expression 1 2 + 3 *.This example demonstrates one of the advantages of postfix—there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write (1 + 2) * 3.

Write a postfix expression that is equivalent to 1 + 2 * 3.

Source: http://openbookproject.net/thinkcs/python/english2e/ch19.html