*12.4 A STACK ADT

An abstract data type is called *abstract* because its description is independent of its implementation. In the epigraph opening this chapter, Barbara Liskov, the pioneering researcher and Turing Award winner who first developed the idea of an abstract data type, explained that "the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant." For example, our understanding of a list as a sequence of objects that may be appended, modified, and searched is independent of whether those objects are stored in sequential blocks of computer memory, archived on a hard drive or written on paper by a barrel of monkeys. The hidden implementation of an ADT is called a ***data structure***. Separating the ADT from the underlying data structure allows us to ignore the implementation details and use abstract data types at a high level to solve problems that would otherwise be out of reach.

An abstract data type that stores a group or sequence of data is known as a ***collection***. We have used several of Python's collection abstract data types—strings, lists, tuples, and dictionaries—and built new types based on these ADTs (e.g., ordered pairs, boids, graphs) to solve specific problems. In this section, we will demonstrate how to define and implement a new collection type by fleshing out the design and implementation of a *stack*.

A stack is a restricted list in which items are inserted and deleted only from one end, called the *top*. When an item is added to a stack, it *pushed* onto the top of the stack. When an item is deleted from a stack, it is *popped* from the same end. Therefore, when an item is popped, it is always the last item that was pushed. For this reason, a stack is known as a "last in, first out" (LIFO) structure.

Stacks have many uses, many of which have to do with reversing some process. (If you completed Project 9.1, you used a stack to draw figures from the strings produced by Lindenmayer systems.) As a simple example, suppose we want to reverse a string. (We used other approaches to do this in Exercises 6.1.14 and 9.2.7.) Using a stack, we can iterate over the characters of the string, pushing them onto the stack one at a time, as illustrated in steps 1–4 in Figure 1. Then, as illustrated in steps 5–8, we can pop the characters and concatenate them to the end of a new string as we do. Because the characters are popped in the reverse order in which they were pushed, the final string is the reverse of the original.

Based on this discussion, the main attribute of the stack ADT is an ordered collection of

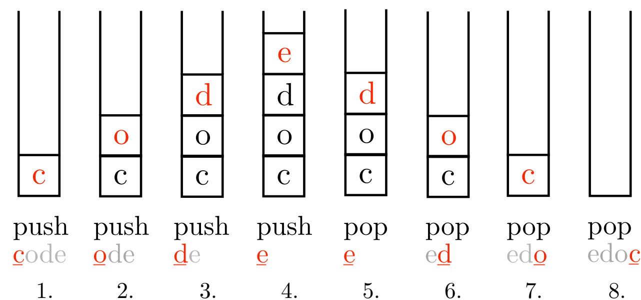

Figure 1 Reversing the string `'code'` with a stack.

Copyright Taylor and Francis, 2021

items in the stack. We also need to know when the stack is empty, so we will also maintain the length of the stack as an attribute.

Instance Variable	Description
items	the items on the stack
length	the number of items on the stack

In addition to the push and pop functions, it is useful to have a function that allows us to peek at the item on the top of the stack without deleting it, and a function that tells us when the stack is empty.

Method	Arguments	Description
create	—	create a new empty Stack instance
top	—	return the item on the top of the stack
pop	—	return and delete the item on the top of the stack
push	an item	insert a new item on the top of the stack
is empty	—	return true if the stack is empty, false otherwise

Stack class

Based on this specification, an implementation of a `Stack` class is fairly straightforward. (To save space, we are using short docstrings based on the ADT specification.)

```
class Stack:
    """A stack class."""

    def __init__(self):
        self._stack = [ ]  # the items in the stack

    def top(self):
        """Return the item on the top of the stack."""

        if len(self._stack) > 0:
            return self._stack[-1]
        raise IndexError('stack is empty')

    def pop(self):
        """Return and delete the item on the top of the stack."""

        if len(self._stack) > 0:
            return self._stack.pop()
        raise IndexError('stack is empty')

    def push(self, item):
        """Insert a new item on the top of the stack."""

        self._stack.append(item)

    def isEmpty(self):
        """Return true if the stack is empty, false otherwise."""

        return len(self._stack) == 0
```

We implement the stack with a list named `self._stack`, and define the end of the list to be the top of the stack. We then use the `append` and `pop` list methods to push and pop items to and from the top of the stack, respectively. (This is the origin of the name for the `pop` method; without an argument, the `pop` method deletes the item from the end of the list.)

Reflection 1 *The stack ADT contained a length attribute. Why do we not need to include this in the class?*

We do not include the length attribute because the list that we use to store the stack items maintains its length for us; to determine the length of the stack internally, we can just refer to `len(self._stack)`. For example, in the `top` method, we need to make sure the stack is not empty before attempting to access the top element. If we tried to return `self._stack[-1]` when `self._stack` is empty, the following error will result:

```
IndexError: list index out of range
```

Likewise, in the `pop` method of `Stack`, if we tried to return `self._stack.pop()` when `self._stack` is empty, we would get this error:

```
IndexError: pop from empty list
```

These are not very helpful error messages, both because they refer to a list instead of the stack and because the first refers to an index, which is not a concept that is relevant to a stack. In the `top` and `pop` methods, we remedy this by using the `raise` statement. You may recall that familiar errors like `IndexError`, `ValueError`, and `SyntaxError` are called *exceptions*. The `raise` statement "raises" an exception which, by default, causes the program to end with an error message. The `raise` statements in the `top` and `pop` methods raise an `IndexError` exception and print a more helpful message after it:

```
IndexError: stack is empty
```

Reflection 2 *If you have not already done so, type the* `Stack` *class into a module named* `stack.py`*. Then run the following program.*

```
import stack

def main():
    myStack = stack.Stack()
    myStack.push('one')
    print(myStack.pop())  # prints "one"
    print(myStack.pop())  # exception

main()
```

Reversing a string

Now let's explore a few ways we can use our new `Stack` class. First, the following function uses a stack to reverse a string, using the algorithm we described above.

```
import stack

def reverse(text):
    """ (docstring omitted) """

    characterStack = stack.Stack()
    for character in text:
        characterStack.push(character)

    reverseText = ''
    while not characterStack.isEmpty():
        character = characterStack.pop()
        reverseText = reverseText + character
    return reverseText
```

Reflection 3 *If the string* `'class'` *is passed in for* `text`*, what is assigned to* `characterStack` *after the* `for` *loop? What string is assigned to* `reverseText` *at the end of the function?*

Converting numbers to other bases

For another example, let's write a function that converts an integer value to its corresponding representation in any other base. In Section 3.2 and Tangent 3.3, we saw that we can represent integers in other bases by simply using a number other than 10 as the base of each positional value in the number. For example, the numbers 234 in base 10, 11101010 in base 2, EA in base 16, and 1414 in base 5 all represent the same value, as shown below. To avoid ambiguity, we use a subscript with the base after any number not in base 10.

$$\begin{aligned}11101010_2 &= 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 128 + 64 + 32 + 0 + 8 + 0 + 2 + 0 \\ &= 234.\end{aligned}$$

$$\begin{aligned}\mathrm{EA}_{16} &= 14 \times 16^1 + 10 \times 16^0 \\ &= 224 + 10 \\ &= 234.\end{aligned}$$

$$\begin{aligned}1414_5 &= 1 \times 5^3 + 4 \times 5^2 + 1 \times 5^1 + 4 \times 5^0 \\ &= 125 + 100 + 5 + 4 \\ &= 234.\end{aligned}$$

To represent an integer value in another base, we can repeatedly divide by the base and add the digit representing the remainder to the front of a growing string. For example, to convert 234 to its base 5 representation:

1. $234 \div 5 = 46$ remainder $\underline{4}$
2. $46 \div 5 = 9$ remainder $\underline{1}$
3. $9 \div 5 = 1$ remainder $\underline{4}$
4. $1 \div 5 = 0$ remainder $\underline{1}$

In each step, we divide the quotient obtained in the previous step (in red) by the base. At the end, the remainders (underlined), in reverse order, comprise the final representation. Since we obtain the digits in the opposite order that they appear in the number, we can push each one onto a stack as we get it. Then, after all of the digits are found, we can pop them off the stack and append them to the end of a string representing the number in the desired base.

```
import stack

def convert(number, base):
    """Represent a non-negative integer value in any base.

    Parameters:
        number: a non-negative integer
        base: the base in which to represent the value

    Return value: a string representing the number in the given base
    """

    digitStack = stack.Stack()
    while number > 0:
        digit = number % base
        digitStack.push(digit)
        number = number // base

    numberString = ''
    while not digitStack.isEmpty():
        digit = digitStack.pop()
        if digit < 10:
            numberString = numberString + str(digit)
        else:
            numberString = numberString + chr(ord('A') + digit - 10)
    return numberString
```

Exercises

12.4.1* Add a `__str__` method to the `Stack` class. Your representation should indicate which end of the stack is the top, and whether the stack is empty.

12.4.2. Enhance the `convert` function so that it also works correctly for negative integers.

12.4.3* Write a function that uses a stack to determine whether a string is a palindrome.

12.4.4. Write a function that uses a stack to determine whether the parentheses in an arithmetic expression (represented as a string) are balanced. For example, the parentheses in the string `'(1+(2+3)*(4 - 5))'` are balanced but the parentheses in the strings `'(1+2+3)*(4-5))'` and `'(1+(2+3)*(4-5)'` are not.

12.4.5. When a function is called recursively, a representation (which includes the values of its local variables) is pushed onto the top of a stack. The use of a stack can be seen in Figures 9.11, 9.12, and 9.15. When a function returns, it is popped

from the top of the stack and execution continues with the function instance that is on the stack below it.

So recursive algorithms can be rewritten iteratively by replacing the recursive calls with an explicit stack. Rewrite the recursive depth-first search function from Section 9.5 iteratively using a stack. Start by creating a stack and pushing the source position onto it. Then, inside a `while` loop that iterates while the stack is not empty, pop a position and check whether it can be visited. If it can, mark it as visited and push the four neighbors of the position onto the stack. The function should return `True` if the destination can be reached, or `False` otherwise.

12.4.6. A ***queue*** abstract data type is like a stack except items are inserted and removed from opposite ends. The insert operation, called enqueue, inserts a new item at the rear of the queue, and the deletion operation, called dequeue, deletes an item from the front of the queue. You may remember using a queue in the breadth-first search algorithm in Section 11.2. Write a class that implements a queue ADT, as defined below.

Instance Variable	Description
items	the items in the queue
length	the number of items in the queue

Method	Arguments	Description
front	—	return the item at the front of the queue
dequeue	—	return and delete the item at the front of the queue
enqueue	an item	insert a new item at the rear of the queue
is empty	—	return true if the queue is empty, false otherwise

12.4.7. Write a class named `Pointset` that contains a collection of points. Inside the class, the points should be stored in a list of `Pair` objects (from Section 12.2). Your class should implement the following methods:

- `insert(self, x, y)` adds a point (x,y) to the point set
- `__len__(self)` returns the number of points (implements the `len` function for a `PointSet`)
- `_distance(self, index1, index2)` returns the distance between the two points with the given indices in the list of points (a private method)
- `centroid(self)` returns the centroid as a `Pair` object (see Section 12.2)
- `closestPoints(self)` returns the two closest points as `Pair` objects (see Exercise 12.2.7)
- `farthestPoints(self)` returns the two farthest points as `Pair` objects
- `diameter(self)` returns the distance between the two farthest points
- `draw(self, tortoise, color)` draws the points with the given turtle and color

Use your class to implement a program that is equivalent to that in Exercise 12.2.7. Your program should initialize a new `PairSet` object (instead of a list), insert each new point into the `PairSet` object, call your `closestPairs`,

`farthestPairs`, and `centroid` methods, draw all of the points with the `draw` method of your `PairSet` class, and draw the closest pair, farthest pair and centroid in different colors using the `draw` method of the `Pair` class (as assigned in that exercise).

Selected Exercise Solutions

12.4.1

```
def __str__(self):
    if len(self) == 0:
        stackString = '| (empty) | '
    else:
        stackString = '| '
        for item in self._stack:
            stackString = stackString + repr(item) + ' | '
    return stackString + '<- top'
```

12.4.3

```
def palindrome(text):
    palStack = stack.Stack()
    for character in text[:len(text) // 2]:
        palStack.push(character)
    for character in text[(len(text) + 1) // 2:]:
        if character != palStack.pop():
            return False
    return True
```