Lesson 32. Working with your own object types

After reading lesson 32, you'll be able to

- Define a class to simulate a stack
- Use a class with other objects you define

At this point, you know how to create a class. Formally, a class represents an object type in Python. Why do you want to make your own object types in the first place? Because an object type packages a set of properties and a set of behaviors in one data structure. With this nicely packaged data structure, you know that all objects that take on this type are consistent in the set of data that defines them, and consistent in the set of operations that they can perform.

The useful idea behind object types is that you can build upon object types you create to make objects that are more complex.

Consider this

Subdivide each of the following objects into smaller objects, and those into smaller objects, until you can define the smallest object by using a built-in type (`int`, `float`, `string`, `bool`):

- Snow
- Forest

Answer:

- Snow is made up of snowflakes. Snowflakes have six sides, and are made up of crystals. Crystals are made up of water molecules arranged in a certain configuration (a list).
- A forest is made up of trees. A tree has a trunk and leaves. A trunk has a length (float) and a diameter (float). Leaves have a color (string).

32.1. DEFINING A STACK OBJECT

In lesson 26, you used lists along with a series of appends and pops to implement a stack of pancakes. As you were doing the operations, you were careful to make sure that the operations were in line with the behavior of a stack: add to the end of the list and remove from the end of the list.

Using classes, you can create a stack object that enforces the stack rules for you so you don't have to keep track of them while the program runs.

Thinking like a programmer

Using a class, you hide implementation details from people using the class. You don't need to spell out how you'll do something, just that you want certain behaviors; for example, in a stack you can add/remove items. The implementation of these behaviors can be done in various ways, and these details aren't necessary to understand what the object is and how to use it.

32.1.1. Choosing data attributes

You name the stack object type `Stack`. The first step is to decide how to represent a stack. In lesson 26, you used a list to simulate the stack, so it makes sense to represent the stack by using one attribute: a list.

Thinking like a programmer

When deciding which data attributes should represent an object type, it may

- Write out which data types you know and whether each would be appropriate to use. Keep in mind that an object type can be represented by more than one data attribute.
- Start with the behavior you'd like the object to have. Often, you can decide on data attributes by noticing that the behaviors you want can be represented by one or more data structures you already know.

You typically define data attributes in the initialization method for your class:

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

The stack will be represented using a list. You can decide that initially a stack is empty, so you initialize a data attribute for the `Stack` object by using `self.stack = []`.

32.1.2. Implementing methods

After deciding on the data attributes that define an object type, you need to decide what behaviors your object type will have. You should decide how you want your object to behave and how others who want to use your class will interact with it.

Listing 32.1 provides the full definition for the `Stack` class. Aside from the initialization method, seven other methods define ways in which you can interact with a stack-type object. The method `get_stack_elements` returns a copy of the data attribute, to prevent users from mutating the data attribute.

The methods `add_one` and `remove_one` are consistent with the behavior of a stack; you add to one end of the list, and you remove from the same end. Similarly, the methods `add_many` and `remove_many` add and remove a certain number of times, from the same end. The method `size` returns the number of items in the stack. Finally, the method `prettyprint_stack` prints (and therefore returns `None`) each item in the stack on a line, with newer items at the top.

Listing 32.1. Definition for the `Stack` class

```
class Stack(object):
    def __init__( self):
        self.stack = []                    1
    def get_stack_elements(self):
        return self.stack.copy()           2
    def add_one(self , item):
        self.stack.append(item)            3
    def add_many(self , item, n):
        for i in range(n):                 4
            self.stack.append(item)        4
    def remove_one(self):
        self.stack.pop()                   5
    def remove_many(self , n):
        for i in range(n):                 6
            self.stack.pop()               6
    def size(self):
        return len(self.stack)             7
    def prettyprint(self):
        for thing in self.stack[::-1]:     8
            print('|_',thing, '_|')        8
```

- ***1*** **A list data attribute defines the stack.**
- ***2*** **Method to return a copy of the data attribute representing the stack**
- ***3*** **Method to add one item to the stack; adds it to the end of the list**
- ***4*** **Method to add n of the same item to the stack**
- ***5*** **Method to remove one item from stack**
- ***6*** **Method to remove n items from the stack**
- ***7*** **Method to tell you the number of items in the stack**
- ***8*** **Method to print a stack with each item on a line, with newer items on top**

One thing is important to note. In the implementation of the stack, you decided to add and remove from the end of the list. An equally valid design decision would have been to add and remove from the beginning of the list. Notice that as long as you're consistent with your decisions and the object's behavior that you're trying to implement, more than one implementation may be possible.

Q1:

Write a method for the `Stack` object, named `add_list`, which takes in a list as a parameter. Each element in the list is added to the stack, with items at the beginning of the list being added to the stack first.

32.2. USING A STACK OBJECT

Now that you've defined a `Stack` object type with a Python class, you can start to make `Stack` objects and do operations with them.

32.2.1. Make a stack of pancakes

You begin by tackling the traditional task of adding pancakes to your stack. Suppose a pancake is defined by a string representing the flavor of pancake: `"chocolate"` or `"blueberry"`.

The first step is to create a stack object to which you'll add your pancakes. Listing 32.2 shows a simple sequence of commands:

- Create an empty stack by initializing a `Stack` object.
- Add one blueberry pancake by calling `add_one` on the stack.
- Add four chocolate pancakes by calling the `add_many` method on the stack.

The items added to the stack are strings to represent the pancake flavors. All methods you call are on the object you created, using dot notation.

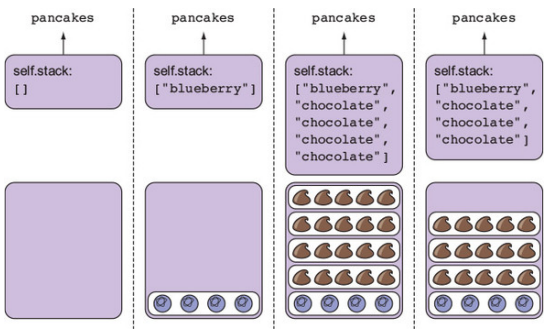
Listing 32.2. Making a `Stack` object and adding pancakes to it

```
pancakes = Stack()                      1
pancakes.add_one("blueberry")           2
pancakes.add_many("chocolate", 4)       3
print(pancakes.size())                  4
pancakes.remove_one()                   5
print(pancakes.size())                  6
pancakes.prettyprint()                  7
```

- ***1* Creates a stack and binds the Stack object to a variable named pancakes**
- ***2* Adds one blueberry pancake**
- ***3* Adds four chocolate pancakes**
- ***4* Prints five**
- ***5* Removes the string that was added last, a "chocolate" pancake**
- ***6* Prints four**
- ***7* Prints each pancake flavor on a line: three chocolate ones on top, and one blueberry at the bottom**

Figure 32.1 shows the steps to adding items to the stack and the value of the list data attribute, accessed by `self.stack`.

Figure 32.1. Starting from the left, the first panel shows an empty stack of pancakes. The second panel shows the stack when you add one item: one `"blueberry"`. The third panel shows the stack after you add four of the same item: four `"chocolate"`. The last panel shows the stack after you removed an item: the last one added, one `"chocolate"`.



Notice that in this code snippet, every method behaves exactly like a function: it takes in parameters, does work by executing commands, and returns a value. You can have methods that don't return an explicit value, such as the `prettyprint` method. In this case, when you call the method, you don't need to print the result because nothing interesting is returned; the method itself prints some values.

32.2.2. Make a stack of circles

Now that you have a `Stack` object, you can add any other type of object to the stack, not just atomic objects (`int`, `float`, or `bool`). You can add ob

You wrote a class that represented a `Circle` object in lesson 31, so now you can create a stack of circles. Listing 32.3 shows you the code to do this. This is similar to the way you added pancakes in listing 32.2. The only difference is that instead of strings representing pancake flavors, you now have to initialize a circle object before adding it to the stack. If you're running the following listing, you'll have to copy the code that defines a `Circle` object into the same file so that Python knows what a `Circle` is.

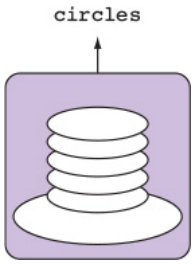
Listing 32.3. Making a `Stack` object and adding `Circle` objects to it

```
circles  = Stack()                    1
one_circle = Circle()                 2
one_circle.change_radius(2)           2
circles.add_one(one_circle)           2

for i in range(5):                    3
    one_circle = Circle()             4
    one_circle.change_radius(1)       4
    circles.add_one(one_circle)       4

print(circles.size())                 5
circles.prettyprint()                 6
```

- ***1*** **Creates a stack and binds the Stack object to a variable named circles**
- ***2*** **Creates a new circle object, sets its radius to 2, and adds the circle to the stack**
- ***3*** **A loop to add five new circle objects**
- ***4*** **Creates a new circle object each time through the loop, sets radius to 1, and adds it to the stack**
- ***5*** **Prints six**
- ***6*** **Prints Python information related to each circle object (its type and location in memory)**

Figure 32.2 shows how the stack of circles might look.

Figure 32.2. The circle with a radius of 2 is at the bottom because it's added first. Then you make a new circle with a radius of 1, five times, and add each one to the stack.



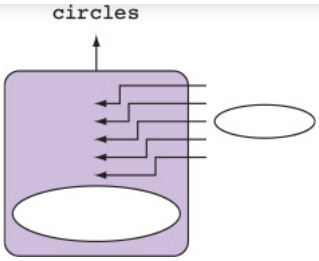
You may also notice that you have a method in the `Stack` class named `add_many`. Instead of a loop that adds one circle at a time, suppose you create one circle with radius 1 and call `add_many` on the stack with this object's properties, as in the following listing, and illustrated in figure 32.3.

Listing 32.4. Making a `Stack` object and adding the same circle object many times

```
circles  = Stack()                    1
one_circle = Circle()                 1
one_circle.change_radius(2)           1
circles.add_one(one_circle)           1

one_circle = Circle()                 2
one_circle.change_radius(1)           2
circles.add_many(one_circle, 5)       3

print(circles.size())                 4
circles.prettyprint()                 5
```

- ***1*** **Same operations as listing 32.3**
- ***2*** **Creates a new circle object; sets its radius to 1**
- ***3*** **Adds the same circle object five times using a method defined in Stack class**
- ***4*** **Prints six, the total number of circles added at this point**
- ***5*** **Prints Python information related to each circle object (its type and location in memory)**

Figure 32.3. The circle with a radius of 2 is at the bottom because it's added first. Then you make one circle with a radius of 1 and add this same circle object five times to the stack



Let's compare how the two stacks look from listings 32.3 and 32.4. In listing 32.3, you created a new circle object each time through the loop. When you output your stack by using the `prettyprint` method, the output looks something like this, representing the type of the object being printed and its location in memory:

```
|_ <__main__.Circle object at 0x00000200B8B90BA8> _|
|_ <__main__.Circle object at 0x00000200B8B90F98> _|
|_ <__main__.Circle object at 0x00000200B8B90EF0> _|
|_ <__main__.Circle object at 0x00000200B8B90710> _|
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x00000200B8B7BF28> _|
```

In listing 32.4, you created only one new circle object and added that object five times. When you output your stack by using the `prettyprint` method, the output now looks something like this:

```
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x00000200B8B7BA58> _|
|_ <__main__.Circle object at 0x000001F1E0E0CA90> _|
```

Using the memory location printed by Python, you can see the difference between these two pieces of code. Listing 32.3 creates a new object each time through the loop and adds it to the stack; it just so happens that each object has the same data associated with it, a radius of 1. On the other hand, listing 32.4 creates one object and adds the same object multiple times.

In lesson 33, you'll see how to write your own method to override the default Python `print` method so that you can print information related to your own objects instead of the memory location.

Quick check 32.2

Q1:

Write code that creates two stacks. To one stack, the code adds three circle objects with radius 3, and to the other it adds five of the exact same rectangle object with width 1 and length 1. Use the classes for `Circle` and `Rectangle` defined in lesson 31.

SUMMARY

In this lesson, my objective was to teach you how to define multiple objects and use them both in the same program. Here are the major takeaways:

- Defining a class requires deciding how to represent it.
- Defining a class also requires deciding how to use it and what methods to implement.
- A class packages properties and behaviors into one object type so that all objects of this type have the same data and methods in common.
- Using the class involves creating one or more objects of that type and performing a sequence of operations with it.

Let's see if you got this...

Q32.1

Write a class for a queue, in a similar way as that for the stack. Recall that items added to a queue are added to one end, and items removed from the queue are removed from the other end:

- Decide which data structure will represent your queue.
- Implement `__init__`.
- Implement methods to get the size, add one, add many, remove one, remove many, and to show the queue.
- Write code to create queue objects and perform some of the

PREV
Lesson 31. Creating a class for an object type

NEXT
Lesson 33. Customizing classes