159.172 Computational Thinking

Chapter 3: Abstract Data Types using Python classes

Making your own objects, and especially types or *classes* of objects, is a central concept in Python - so central in fact that Python is called an *object-oriented* language. You will have learned the basics about classes in 159171. If you need a refresher on the basics of Python classes, go to Chapter 13 of Paul Craven's introductory Python online text.

In this chapter, we review Python classes, in particular concentrating on the aspects of classes that make them suited to implementing *abstract data types* , *ADT's* for short.

Data abstraction is the separation of the properties of a data type (its values and operations) from the implementation of that data type. 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 specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- 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.
- 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.
- Well-known ADTs, such as the Stack ADT introduced later in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
- 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. Client code interacts with instances of an ADT by invoking one of the operations defined by its *interface*. The set

of operations can be grouped into four categories:

1. *Constructors* : create and initialize new instances of the ADT.
2. *Accessors* : return data contained in an ADT instance without modifying it.
3. *Mutators* : modify the contents of an ADT instance.
4. *Iterators* : process individual data components of an ADT instance sequentially.

The advantages of working with abstract data types all boil down to the idea of focussing on the *what* rather than the *how*. We now look at the concepts that make Python classes suited to implementing ADT's.

3.1 Polymorphism

Back to top

You have already seen polymorphism in action when you have used the same operation on objects from different classes and it has worked as you expected. This means that even if you don't know what kind of object a variable refers to, you may still be able to perform an operation on that variable and the operation will work differently depending on the class of the object.

Examples:

```
>>> 2+3
5

>>> 'my'+'string'
'mystring'

>>> [1, 2, 3]+['a', 'b', 'c']
[1, 2, 3, 'a', 'b', 'c']
```

Polymorphism is at work every time you can do something to an object without knowing exactly what kind of object it is. In the example above, the plus operator (+) works for numbers, strings and lists. In each case, it works in a way that is appropriate for the class of the objects to which it is applied. The arguments can be *anything that supports addition*, more precisely, the arguments can be from any class that supports addition of pairs of instances.

More examples:

```
>>> x = [1, 2, 3]
>>> y = ['a', 'b', 'c']
x+y
[1, 2, 3, 'a', 'b', 'c']

>>> x = 5
>>> x+y
Traceback (most recent call last):
  File "", line 1, in
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Many built-in Python operators and functions are polymorphic. When you use these in your

own function and method definitions you will find that your own definitions are polymorphic by default. For example, if you want to write a function that prints a message about the length of an object, all that is required is that it *has* a length, that is, that the *len* function will work on it.

```
def length_message(x):
    print "The length of", repr(x), "is", len(x)

>>> length_message('string')
The length of 'string' is 6

>>> length_message([1,2,3,4])
The length of [1, 2, 3, 4] is 4

>>> length_message([(1,2), (5,6), (8,9,10)])
The length of [(1, 2), (5, 6), (8, 9, 10)] is 3
```

Polymorphism is an *abstraction* mechanism. It can help you to deal with components of your program without caring about unnecessary detail. You can use an object without knowing its class, all you need to know is that it supports the operation that you want.

3.2 Encapsulation

Back to top

Encapsulation enables you to use an object without worrying about how it's constructed. Objects may hide (encapsulate) their internal state. In some languages, this means that access to the state (attributes) of an object is available only through their methods. This is the behavior that we want when implementing ADT's. Remember, we want to ensure that a program interacts with instances of an ADT by invoking one of the operations defined by its *interface*, not via any other mechanism.

In Python, by default, you can access the attributes of an object "from the outside". To make a method or an attribute private (inaccessible from the outside) you just need to start its name with two underscores. The method or attribute can still be used inside the class as usual, but is not accessible (directly) from outside the class. Attributes that are private are accessible only through *accessor* methods. If you use a single underscore to start the name of an attribute or method, then it should be *regarded* as private, although this is not enforced. Names with a single initial underscore are not imported with starred imports:

```
(from module import *)
```

Example:

```
class Person:

    __surname = 'Allen'

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name
```

```
    def __secretmessage(self):
        print 'I should not tell you my name, it is ' + self.name + '
' + self.__surname

    def _semi_secret(self):
        print 'I have not told you my name.'

    def public_message(self):
        print 'The secret message is: '
        self.__secretmessage()

x = Person()
y = Person()
x.setName('John')
y.setName('Fred')

x.public_message()

y._semi_secret()

y.__secretmessage()
```

Running the code above gives the following output:

```
The secret message is:
I should not tell you my name, it is John Allen

I have not told you my name.

Traceback (most recent call last):
  File "/Users/cmmccart/Documents/workspace/tester/testfile.py", line
25, in
    y.__secretmessage()
AttributeError: Person instance has no attribute '__secretmessage'
```

If we add the code

```
   def getSurname(self):
       return self.__surname

print y.getName()
print y.getSurname()
print y.__surname()
```

we get:

```
Fred
Allen
Traceback (most recent call last):
  File "/Users/cmmccart/Documents/workspace/tester/testfile.py", line
28, in
    print y.__surname()
AttributeError: Person instance has no attribute '__surname'
```

As this example demonstrates, in order to access private attributes and methods in Python, you may use accessor methods, such as *getSurname()* or methods like *public_message()* that

invoke private attributes or methods.

3.3 Inheritance

Back to top

Inheritance allows us to build classes that are "specialisations" of other classes, in other words, that inherit all of the attributes and methods of another class and then expand on these. A class may be the subclass of one or more other classes, but *multiple* inheritance (inheriting from more than one class) is something that you should use sparingly, it can lead to some messy complications. A subclass will expand on the definitions in its superclasses and may *override* some of them. You indicate the superclass in a *class* statement by writing it in parentheses after the class name. If you are using multiple inheritance then you need to be careful about the order of the superclasses in the class statement, since the methods in the earlier classes override the methods in the later ones (if you have two different methods with the same name.)

```
class Filter:
    def __init__(self):
        self.blocked = []

    def filter(self, sequence):
        return [x for x in sequence if x not in self.blocked]

class SPAMFilter(Filter):   # SPAMFilter is a subclass of Filter
    def __init__(self):
        self.blocked = ['SPAM']

>>> f = Filter()
>>> f.filter([1, 2, 3])
[1, 2, 3]

>>> s = SPAMFilter()
>>> s.filter( ['SPAM', 'SPAM', 'eggs', 'bacon', 'SPAM', 'SPAM'])
['eggs', 'bacon']
```

The definition of __init__ from the Filter class is *overridden* in the definition of the SPAMFilter class, by simply providing a new definition. The definition of the filter method is inherited from the Filter class and carries over to the SPAMFilter class, you don't need to write the definition again.

3.4 Constructors

Back to top

All classes in Python should define a special method called the *constructor*, which defines and initializes the data to be contained in the object. The contractor is automatically called when an instance of the object is created. We create a constructor in Python using the __init__ method.

```
class Point:
    def __init__(self, x, y):
        self.xCoord = x
        self.yCoord = y
```

An instance of a class object is created by invoking the constructor, which is done by specifying the class name along with any required arguments, just like a function call.

```
>>> pointA = Point(5,7)
>>> pointB = Point(0,0)
```

You can see that the constructor is defined with three parameters but we supplied only two arguments when creating the objects. *self* is the special parameter that must be included in each method definition and it must be listed first. When any method, including the constructor method, is called, the *self* parameter is automatically bound to the object on which the method is called.

You sometimes need to be careful when overriding the constructor method of a superclass by creating a constructor for a subclass. You may need to call the constructor of the superclass or risk having an object that isn't properly initialized.

```
class Bird:
    def __init__(self):
        self.hungry = True

    def eat(self):
        if self.hungry:
           print 'Aaaahh…'
        else:
           print 'No thanks!'

class SongBird(Bird):   # SongBird is a subclass of Bird
    def __init__(self):
        self.sound = 'Squwark!'

    def sing(self):
        print self.sound

>>> sb = SongBird()

>>> sb.sing()
Squwark!

>>> sb.eat()   ??????
```

You can fix this problem two ways, by calling the constructor method of the superclass and passing the current "self" as its parameter, or by using the *super* function and passing the current class and current self as arguments.

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squwark!'

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = 'Squwark!'
```

3.5 Iterators

Back to top

Recall that the set of operations we would like to be able to define for ADT's can be grouped into four categories:

1. *Constructors* : create and initialize new instances of the ADT.
2. *Accessors* : return data contained in an ADT instance without modifying it.
3. *Mutators* : modify the contents of an ADT instance.
4. *Iterators* : process individual data components of an ADT instance sequentially.

We now know about constructors, and we could argue that accessors and mutators can be implemented using "regular" methods. What about iterators? We have seen how to iterate over a Python list using a *for* loop. You can iterate over other objects too, as long as their class definition implements the __iter__ method. The __iter__ method returns an *iterator* which is an object with a method called next, callable without any arguments. When you call the next method, the iterator should return its "next" value. If there are no more values to return, the next method will raise a StopIteration exception.

Here is an iterator that returns the sequence of Fibonacci numbers, one by one.

```
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1

    def next(self):
        self.a, self.b = self.b, self.a+self.b
        return self.a

    def __iter__(self):
        return self
```

Here is an iterator that returns the rows of Pascal's triangle, one by one.

```
class Pascal:
    def __init__(self):
        self.lastRow = []
        self.nextRow = [1]

    def next(self):
        self.lastRow = self.nextRow
        self.nextRow = [(a+b) for a,b in
zip([0]+self.lastRow,self.lastRow+[0])]!
        return self.lastRow

    def __iter__(self):
        return self
```

If we run the following:

```
pascal = Pascal()
for row in pascal:
    print row
```

```
        if len(row) > 12:
            break
```

We get the output:

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
[1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1]
[1, 11, 55, 165, 330, 462, 462, 330, 165, 55, 11, 1]
[1, 12, 66, 220, 495, 792, 924, 792, 495, 220, 66, 12, 1]
```

In many cases, you put the definition for the iterator into a *separate class* and then have the __iter__ method in your ADT class call an instance of the iterator.

3.6 The Stack ADT

Back to top

In this section, 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. Another collection that we have seen is the Python list.

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

- **__init__()**
  Initialize a new empty stack.
- **push(new_item)**
  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.
- **isEmpty()**
  Check whether the stack is empty.

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

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 the 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 isEmpty(self) :
      return (self.items == [])
```

- A Stack object contains an attribute named *items* that is a list of items in the stack. The constructor method sets items to the empty list.
- To push a new item onto the stack, push appends it onto items, making it the last item in the list.
- To pop an item off the stack, pop uses the list method pop to remove and return the last item on the list.
- Finally, to check if the stack is empty, isEmpty compares items to the empty list.

Suppose we want to add some more operations to our Stack ADT:

- **top()**
  Return the item on top of the stack, without removing it.
- **peek()**
  Return the item that is directly below the top item, without removing it.
- **pop_many(n)**
  Remove and return the top n items from the stack. If less than n elements are on the stack, the tuple will contain all stack entries and the stack will then be empty again.
- **push_many(seq)**
  Push the objects in seq from left to right onto the stack.

Consider how you could add these operations to the Python list implementation that we have built here.

Back to top