Object-Oriented Programming
Statistics 650/750
Week 7 Tuesday

Alex Reinhart and Christopher Genovese

10 Oct 2017

Announcements

- Lots of questions about "command-line drivers" requested in homeworks. All this means is this: we want to be able to run your tests by running a command like

```
Rscript cow_proximity_tests.R
```

  directly from the command line.

- Remember that your projects are currently under way and mandatory peer review will be **next Thursday**, October 19, in class. Talk to us if you have questions!

Object-oriented programming

What is object-oriented programming?

We are used to programming with *verbs* – functions. We define a bunch of functions to handle different parts of our problem, then fit the pieces together.

Sometimes it's more useful to program with *nouns* – pieces of data with specific behaviors. An *object* is exactly that: a collection of data with defined *methods* which operate on that data. By choosing our objects carefully and defining interfaces for their behaviors, they can make our code more generalizable while limiting the spread of complexity.

Good object-oriented code improves development like interchangeable parts improve manufacturing.

A forest of trees

Let's start with an example. I'd like to create a binary tree, as we discussed a couple weeks ago. (The example would work equally well with any of the data structures we discussed.)

One way would be to define the tree recursively, like we did last week, and pass around a pointer to the root node. Different functions operating on that tree would manipulate it manually, by recursing through the tree and performing their operations.

But this is not general. If I want to switch to a different kind of tree – say, a red-black tree instead of an ordinary binary tree – all my code needs to be updated. If I change the representation of my data, using arrays to store node values and connections instead of a recursive structure, all my code needs to be updated. The complexity of my implementation has leaked into the rest of my program, and everything is specialized to this specific implementation.

Classes, attributes, and methods

I can create a *class* specifying a binary tree:

```
class BinaryTree:
    def __init__(self, nodes):
        """Construct a binary tree."""

        self.value = nodes.pop()
        for node in nodes:
            self.insert(node)

    def insert(self, value):
        """Insert a node into the tree."""

        if value < self.value:
            self.left.insert(value)
        else:
            ...

    def delete(self, value):
        ...

    def search(self, value):
        ...

    def inorder(self):
        """Iterate over the tree in sorted order."""

        ...
```

The class specifies a *constructor* (`__init__`) which creates a binary tree and the *methods* that operate upon it. Note that each method takes the parameter `self`. When I call `delete` on a binary tree object, the `delete` method gets that object as `self` so it may operate upon it.

Then I can create and use binary trees:

```
# calls the __init__ method to construct a tree
b = BinaryTree([2, 4, 7, 1, 16, 3])

b.search(4) # True
b.insert(5)
b.search(5) # True

b2 = BinaryTree([-3, 2, 17])
b2.search(5) # False
```

Here `b` and `b2` are *instances* of the `BinaryTree` class. Each has separate data and does not affect the other.

The *attributes* of objects are also available:

```
b.left  # the left child
```

These can be manipulated by other code, though it's often best to make changes through methods instead of direct access.

Interchangeable trees

I want to change my binary tree implementation. I'd like to use a red-black tree, which automatically rebalances the tree so searching always takes O(log n) time. This requires changing the insertion and deletion operations, but a red-black tree is still a binary tree, so searching behaves the same way.

Classes can *inherit* methods and data from other classes:

```
    class RedBlackTree(BinaryTree):
        # No need to define __init__ or search -- they're inherited from BinaryTree

        def insert(self, value):  # these override the methods from BinaryTree
            ...

        def delete(self, value):
            ...
```

Now `RedBlackTree` is a *subclass* of `BinaryTree`. If I call `insert` on a `RedBlackTree` instance, I get the method defined specifically for it; if there is no method defined for it, I get the parent class's method.

Some languages allow *multiple inheritance*: a class can inherit from several classes simultaneously. This should be used with caution. It is often difficult to figure out which of several inherited methods are being used, if the parent classes happen to have methods of the same name. Conflicts can cause great confusion.

There is a relationship between superclasses and subclasses:

```
b = RedBlackTree([1, 2, 4])

isinstance(b, BinaryTree)            # True
isinstance(b, RedBlackTree)          # True
issubclass(RedBlackTree, BinaryTree) # True
```

Because a `RedBlackTree` *is a* `BinaryTree`, any code that expects a `BinaryTree` can work just as well with a `RedBlackTree`, or an `AVLTree`, or whatever other tree types I might define. I can swap them out as necessary. The implementation details of the tree are hidden within the class, and code outside it does not need to know.

Encapsulation

Objects provide *encapsulation*: the data and methods for a single object are wrapped up in one place. All the complexity of the implementation of that object is contained, and other code need only interact through it with its *interface*.

This can limit bugs. If other code tries to directly manipulate the tree instead of calling `insert`, it may accidentally break the rules of the red-black tree or leave the object in an inconsistent state. By only operating through the public interface (the methods), we ensure that the object is always valid and its invariants are maintained.

(Some languages, like Java and C++, allow more formal enforcement of this by marking some data and methods as *private*, only accessible to methods of that class. Outside code cannot see or modify private class data.)

More importantly, this lets me change implementation details of `BinaryTree` without changing any of the code that uses trees. If I want to change how it stores trees, its traversal strategies, and so on, I don't have edit any other code.

SOLID design

Single responsibility Each class should have a single responsibility

Open/closed Open for extension, closed for modification. Write classes that can easily be extended without being modified.

Liskov substitution You should be able to substitute a subclass for a class without breaking the program

Interface segregation Specific interfaces for specific tasks are usually better than one giant interface to do everything for all users

Dependency inversion Depend on the abstractions, not the concrete details. Some languages make this explicit with abstract classes and interfaces

Examples of object usage

Everything

In some languages, *everything* is an object. In Python, for example, lists, dictionaries, strings and so on all have methods, like `foo.sort()`. A great deal of syntax is just sugar for method calls:

```
    class defaultdict(dict):
        def __init__(self, default):
            self.default = default

        def __getitem__(self, key):
            if not key in self:
                self[key] = self.default
            return super(defaultdict, self).__getitem__(key)

    foo = defaultdict(14)
    foo[142] # 14
    foo[142] = 7
    ...
```

Square brackets turn into a `__getitem__` call, which can implement arbitrary behavior. (Don't get too creative – it should match the usual interface of indexing.) Even accesses to instance data (e.g. `foo.bar`) turn into calls to `__getattr__` if the attribute isn't found. See Python's data model documentation.

(A more robust defaultdict is provided in the `collections` module.)

Iterators

Iterators are a generic interface to iterating over sequences. In Python, iterators are implemented using objects with just two methods:

__iter__ Returns the iterator object

__next__ Returns the next item from the sequence. Raises the `StopIteration` exception if there are no more elements.

`__next__` can do arbitrarily complicated calculations, so we could define an iterator which produces elements from any sequence we'd like. Most commonly, iterators iterate over collections.

For example, our binary tree could produce preorder, inorder, and postorder iterators. A dictionary could provide iterators over its keys and values. An infinite sequence could be represented as an iterator – the iterator returns one element at a time, so it need not fit in memory. A file object can produce an iterator over lines or characters in the file. A database query can return an iterator which produces the query results.

Here's a very simple iterator object:

```
class Naturals:
    def __init__(self):
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.count += 1
        return self.count
```

`for` loops automatically support iterators, so we could write:

```
    for element in Naturals():
        do_stuff(element)
```

If functions accept iterators and return iterators, they can be chained:

```
fn main() {
    let input = 1..10;
    let output = input
        .filter(|&item| item % 2 == 0)  // keep even numbers
        .map(|item| item * 2)           // multiply by two
        .fold(0, |accumulator, item| accumulator + item); // sum

    println!("{}", output);
}
```

This only requires a single loop through the input, instead of three.

(Rust example from `http://hoverbear.org/2015/05/02/a-journey-into-iterators/`)

Exceptions and conditions

In Python, exceptions are objects. In R, conditions are S3 objects. You can define new types of exceptions and conditions by defining new classes:

```
    class InputError(Exception):
        def __init__(self, expr, msg):
            self.expr = expr
            self.msg = msg
```

S3 objects in R are just lists with a class attribute, so we can create a constructor:

```
    input_error <- function(text) {
        msg <- paste0("Input error: ", text)

        structure(
            list(message=msg, text=text, call=sys.call(-1)),
            class=c("input_error", "error", "condition")
        )
    }
```

The `structure` function takes its argument and adds the supplied attributes to it.

S3 classes in R

R uses a rather different system from Python or Java for its object-oriented programming. You've probably interacted with this system without even knowing it.

R actually has three (or more!) different systems for object-oriented programming: S3, S4, and RC classes are the biggest. S3 classes are what you'll encounter most often.

S3 classes don't have formal definitions, like in Python, where we state what methods and data are supported by every instance of the class. To make a new instance of a class, we take some object (a list, say, containing the data) and set its `class` attribute:

```
node <- structure(list(left=stuff, right=other_stuff, value=7),
                  class="tree")

## or
node <- list(...)
class(node) <- "tree"
```

The `class` function returns the class of any variable:

```
> class(iris)
[1] "data.frame"
```

Generic functions

Notice we didn't specify the methods of our `tree` class above. Instead, R uses *generic functions*: functions which behave differently for different classes of arguments.

Many built-in functions are generic. For example, if I ask R for the `print` function:

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x7fc3a14eb428>
<environment: namespace:base>
```

`UseMethod` looks up which `print` method to call, based on what class `x` is. What are the options?

```
> methods(print)
  [1] print.acf*
  [2] print.anova*
  [3] print.aov*
  [4] print.aovlist*
  [5] print.ar*
  [6] print.Arima*
  [7] print.arima0*
  [8] print.AsIs
  [9] print.aspell*
 [10] print.aspell_inspect_context*
 [11] print.bibentry*
  ...
[189] print.xtabs*
```

To create a method for your new class, just use the right name:

```
print.tree <- function(tree) {
    ...
}

print(t)
```

If you're creating a brand-new function and want it to be generic, just use `UseMethod`:

```
## we add the ... argument so methods can take more arguments
## if desired
inorder <- function(x, ...) {
    UseMethod("inorder")
}

inorder.binarytree <- function(x) {
```

```
    ## do some stuff
}

inorder.redblacktree <- function(x) {
    ## do different stuff
}

## Called for classes without an inorder method
## otherwise defined
inorder.default <- function(x) {
    ## do stuff
}
```

Model fit objects

In R, the results of most model fits are objects containing slots (list entries) for the data, parameters, diagnostics, and so on. Methods (like `confint` and `plot`) are defined to operate on these objects.

If you fit a new kind of model, you can easily implement the same methods so your results can be manipulated the same way.

The hyperreal numbers

S3 methods are *single-dispatch*: based on the class of the first argument to the method, R figures out which method you want to call. But this isn't the only way to do it.

Suppose I would like to model the *hyperreal numbers*. I won't go into great detail, but the hyperreals offer a rigorous definition of infinitesimals, the "dx"s you often handwaved away in introductory calculus. A hyperreal number has a real part (or standard part) and an infinitesimal part.

So we may define an S4 class in R:

```
setClass("hyperreal", slots=c(x="numeric", dx="numeric"))
```

This states that the class "hyperreal" has two *slots*: x and dx. Slots contain data; each instance of a hyperreal can have different values in those slots.

I can define a function to create a hyperreal from its real ("standard") and infinitesimal parts:

```
    hyper <- function(x, dx) {
        new("hyperreal", x=x, dx=dx)
    }
```

The `new` function constructs a new instance of an object. Slots can be accessed with the @ operator: `foo@dx`.

Methods and multiple dispatch

S4 uses *multiple dispatch*. You can define *generic functions*: functions which behave differently depending on the types of their arguments. Instead of depending on the type of one object, like in our Python or S3 code, they can depend on the types of as many arguments as you'd like. Julia uses a similar system.

For example, to define hyperreal arithmetic, we might do

```
    setMethod("+", signature(e1="hyperreal", e2="hyperreal"),
              function(e1, e2) {
                  hyper(e1@x + e2@x, e1@dx + e2@dx)
              })

    setMethod("+", signature(e1="hyperreal", e2="numeric"),
              function(e1, e2) {
                  hyper(e1@x + e2, e1@dx)
              })
    ...
```

By specifying a *signature*, I'm saying that if + is called with *two* hyperreals, I add them one way, but if it's called with a hyperreal and an ordinary number, I add them a different way. I can create as many different methods as I'd like. I can also handle other mathematical functions on hyperreals:

```
    setMethod("sin", signature(x="hyperreal"),
              function(x) {
                  hyper(sin(x@x), cos(x@x) * x@dx)
              })

    setMethod("cos", signature(x="hyperreal"),
              function(x) {
                  hyper(cos(x@x), - sin(x@x) * x@dx)
              })
    ...
```

Polymorphism

Once I have defined these methods, any functions which work on numerics also work on hyperreals:

```
foo <- function(x) {
    sin(x)^2 + 3*x^2 + log(x) - 4
}
```

foo is *polymorphic* or *generic*: it operates on any type which implements the required operations. Then I have

```
> foo(4)
[1] 45.95904
> foo(hyper(4, 1))
An object of class "hyperreal"
Slot "x":
[1] 45.95904

Slot "dx":
[1] 25.23936
```

It just so happens that 25.23936 is the exact numerical derivative of `foo` when evaluated at x=4. By defining a new object and methods upon it, I can get *exact* numerical derivatives of any function which uses ordinary arithmetic and mathematical functions. (Notice that this is not the secant method or any other approximation method.)

R OOP summary

R has three different systems for object-oriented programming:

S3 The oldest and simplest system, built on lists (usually). An object is just a variable that's been labeled as having a certain class. Generic functions can be written to operate on different classes. Commonly used in base R.

S4 A more sophisticated system with inheritance, multiple dispatch, and more formality. (I used S4 to define the hyperreals.) Less common, but used when needed, such as in the Matrix package.

RC "Reference classes" behave more like objects in Python or Java, with methods called as `object$method(foo)`. RC objects are passed *by reference*, meaning that they are not copied on modification like most R types.

Usually you will use and interact with S3 classes. They're great when you have some objects – like model fits, estimates, data tables, data structures, or whatever – that have common operations defined for them, and should be easily passed to functions which don't care about their internal implementation.

Resources

Principles

- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides. Based on C++, but widely applicable.
- *Refactoring: Improving the Design of Existing Code*, by Fowler, on improving and redesigning object-oriented code.
- *Growing Object-Oriented Software, Guided by Tests*, by Freeman and Pyrce, on test-driven development for object-oriented programs.
- *Software Architecture in Practice*, by Bass, Clements, and Kazman.

Implementation

- For R, Hadley Wickham's *Advanced R* has a detailed chapter on OOP, but does not describe when or why it is useful.
- John Chambers' *Programming with Data* is a detailed introduction to S4, with many examples of its use. (Written originally for S instead of R. This book introduced S4 classes to the world.)
- The Python tutorial has a long section on classes.