Lect. 15: Object-oriented programming in R

STAT598z: Intro. to computing for statistics

Vinayak Rao

Department of Statistics, Purdue University

In [ ]:
```
options(repr.plot.width=5, repr.plot.height=3)
```

functions: an abstraction to encourage modular code:

- Reusables block of code
- Define operations to apply to any input

Object-oriented programming is another kind of abstraction

Object-oriented programming is another kind of abstraction:

- *Encapsulation*: Pack data and functions into classes
- *Polymorphism*: Same functions act differently across classes
- *Inheritence*: Write child classes without copying parent classes

Components of OOP:

Classes: A template for an object (e.g. `purdue`)

- Defines 'properties' of objects (e.g. `name`, `puid`)

Objects An instance of a class (e.g. `varao`)

- Values assigned to properties (`name = 'vinayak'`)

Methods Functions aware of properties of the object

- (e.g. `isFaculty()`)

Why object oriented programming (OOP)?

1) Useful to group variables together

- An object is basically a list, with the `class` attribute set
- A *constructor* creates objects of a class

In [ ]:
```
new_purdue <- function(name, puid, employee ) {
  obj <- list(name = name, puid = puid, employee = employee)
  class(obj) <- 'purdue'; return(obj)
}
varao <- new_purdue('Vinayak', 1234, 'faculty' )
print(varao)
```

Why object oriented programming (OOP)?

2) Tying methods to objects:

- Increase capability of software without increasing complexity for user (*Chambers*): e.g. `print` vs `printMatrix`
- Protects users from implementation details. User only needs to know an interface, and doesn't care about insides. (E.g. `varao$employee =='faculty'` vs `isFaculty(varao)`)

Object oriented (OO) systems in R

R has three OO systems:

- S3: most common OO system in R
- S4: like S3, but more formal
- Reference classes (RC): new, and like OO in other languages

We will concentrate on S3

Suppose `varao` is an object of class `purdue`

Can write a function `print.purdue()` and call when needed

Simpler/clearer to just use `print()`

Two OO paradigms:

Methods in classes

- Would look like `varao.print()`
- C++, python, java, (also the RC system in R)
- methods are 'attached' to objects

Generic functions

- Would look like `print(varao)`
- The S3 and S4 systems in R
- Define method `print.purdue()` but call `print()`
- `print` is a **generic** function that **dispatches** methods

In most OOP languages, methods belong to objects

In R, methods belong to generic functions

- uses `UseMethod()` to call method based on object class

`methods` gives you all methods associated with a generic

In [ ]:
```
methods(print)
```

In [ ]:
```
methods('[')
```

Can also give all methods associated with a class

In [ ]:
```
methods(class= 'matrix')
```

`ftype()` can tell generics from methods

In [ ]:
```
library('pryr')
ftype(print)
ftype(print.data.frame)
```

Why do we need language support for OOP?

Can't we just modify if conditions inside print ?

- Don't want to have to change R code for e.g. print

R's OOP support allows

- extending functionality without touching existing code
- fewer bugs

The S3 system

S3 can be viewed as a naming convention:

- methods look like `generic.class()`
- e.g. `print.table` accessed via the generic `print`

`print(varao)` will

- look for `print.purdue()`
- If no such function, will call `print.default()`

In [ ]:
```
print(varao)
```

In [ ]:
```
print.purdue <- function(x) {
  cat(x$name, ' (PuID:' , x$puid,') is ', x$employee,
      ' at Purdue\n' )
}
print(varao)
```

Inheritence

- An object need not be assigned to just one class
- Classes are from most to least specific

In [ ]:
```
ab12 <- list(name = 'Alice' , puid = '12345' ,
             employee = 'TA' , gpa = 3.8)
class(ab12) <- c('grad' , 'purdue')
print(ab12)
```

In [ ]:
```
inherits(ab12, 'purdue' )
```

In [ ]:
```
gpa.grad <- function(x) print(x$gpa)
#gpa(ab12) # We don ' t have a generic yet!
gpa <- function(x) UseMethod('gpa')
class(ab12)
gpa(ab12)
```

In [ ]:
```
print(ab12)
```

Can also reuse methods using `NextMethod()`

In [ ]:
```
print.grad <- function(x) {
  NextMethod(print)  # calls print.purdue
  cat( ' \n Has GPA ' , x$gpa, '\n')
}
```

In [ ]:
```
print(ab12)
print(varao)
```

Writing generic functions

We've seen how to write methods

To write a generic use `UseMethod()`

```
gpa <- function(x) UseMethod('gpa')
```

Essentially creates vector:

```
paste0('gpa.',c(class(x), default)
```

Searches from left to right for function that exists

If it finds one, calls it, else returns error

Example

Imagine a vector that you wanted to always view backwards

- A stack where new jobs are added to the top

You want to hide from the user that it's stored forwards

In [ ]:
```
my_path <- c('right turn', 'cross street', 'climb stairs')
class(my_path) <- 'stack'
print(my_path)
```

In [ ]:
```
print.stack <- function(x) print(rev(x))
print(my_path) # Are you surprised this works?
```

In [ ]:
```
'[.stack' <- function(x,i) {
  class(x) <- NULL   # why do we need this?
  x[length(x)+1-i]
}
# warning: this messes up your previous print function
```

In [ ]:
```
my_path[3]
```

Object oriented programming

A powerful way to organize software

Allows you to build on existing software without changing it

Can avoid a bewildering set of new names for a generic task

S3 is a very informal system with no real checks

Can assign any class to any object

Can cause trouble if you're not careful