Composing functions and sequencing operations are core programming concepts.

Some notable realizations of sequencing or pipelining operations include:

* Unix’s [|-pipe](https://en.wikipedia.org/wiki/Pipeline_(Unix))
* [CMS Pipelines](https://en.wikipedia.org/wiki/CMS_Pipelines).
* F#‘s forward pipe operator |>.
* Haskel’s Data.Function & operator.
* The [R](https://www.r-project.org/) [magrittr](https://cran.r-project.org/package=magrittr) forward pipe.
* [Scikit-learn](https://scikit-learn.org/stable/index.html)‘s [sklearn.pipeline.Pipeline](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html).

The idea is: many important calculations can be considered as a sequence of transforms applied to a data set. Each step may be a function taking many arguments. It is often the case that only one of each function’s arguments is primary, and the rest are parameters. For data science applications this is particularly common, so having convenient pipeline notation can be a plus. An example of a non-trivial data processing pipeline can be found [here](https://github.com/WinVector/vtreat/blob/master/extras/ModelingPipelines.md).

In this note we will discuss the advanced [R](https://www.r-project.org/) pipeline operator ["dot arrow pipe"](https://winvector.github.io/wrapr/reference/dot_arrow.html) and an S4 class ([wrapr::UnaryFn](https://winvector.github.io/wrapr/index.html" \t "_blank)) that makes working with pipeline notation much more powerful and much easier.

The ideas are:

* The [wrapr](https://github.com/WinVector/wrapr) [dot arrow pipe](https://winvector.github.io/wrapr/reference/dot_arrow.html) includes a detailed S3/S4 configurable interface (detailed in the RJournal [here](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html)). These interfaces are able to treat objects as functions: i.e. they can pipe data into objects.
* The wrapr::UnaryFn class supplies a convenient tool for the [partial function application](https://en.wikipedia.org/wiki/Partial_application) needed to work with pipelines.

Or: pipe notation assumes a world data transforms are single argument functions (with other parameters already bound in), and the UnaryFn derived classes we discuss here help realize such a world.

This can be made clearer with examples.

Suppose we build a linear model of log(y) as follows.

d <- data.frame(

x = c(1, 2, 3, 4, 5),

y = c(3, 5, 20, 50, 150))

model <- lm(log(y) ~ x, data = d)

We can see our predictions in original y-units by making the prediction and then exponentiation:

exp(predict(model, newdata = d))

## 1 2 3 4 5

## 2.459509 6.770839 18.639596 51.313366 141.261725

It is natural to want to apply a model later to new data. This can be done as follows.

d2 <- data.frame(x = 3:7)

exp(predict(model, newdata = d2))

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

The wrapr package allows us to use a "piping into a function" notation as follows.

library("wrapr")

model\_f <- function(df) {

exp(predict(model, newdata = df))

}

d2 %.>% model\_f

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

In the above example the model contents are captured in the function closure. However, it is better practice to explicitly store data in objects.

wrapr supplies a method to do this, which we will now demonstrate.

model\_o <-

fnlist(

pkgfn(

"stats::predict.lm",

arg\_name = "newdata",

args = list(object = model)),

pkgfn(

"exp",

arg\_name = "x"))

cat(format(model\_o))

## UnaryFnList(

## stats::predict.lm(newdata=., object),

## base::exp(x=., ))

Notice model\_o is an object (not a function). However we can pipe into model\_o as if it were a function.

d2 %.>% model\_o

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

This works because model\_o is derived from the S4 class UnaryFn and wrapr has definitions for apply\_right.UnaryFn and apply\_left.UnaryFn, which integrate this class into the [wrapr dot-arrow pipe](https://winvector.github.io/wrapr/reference/dot_arrow.html). The family of UnaryFn classes single argument functions. This system happens to be implemented by wrapr, but wrapr dot arrow extension mechanisms also allow users to build their own pipe-compatible systems. (S3/S4 extension details can be found in the RJournal [here](https://journal.r-project.org/archive/2018/RJ-2018-042/index.html).

The pipe notation is not strictly required as the apply is done through the S4 method wrapr::ApplyTo().

ApplyTo(model\_o, d2)

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

The above methods can be used to wrap substantial functions such as [vtreat::prepare()](https://winvector.github.io/vtreat/reference/prepare.html) to create very powerful data processing pipelines. A more involved example of trying this technique can be found [here](https://github.com/WinVector/vtreat/blob/master/extras/ModelingPipelines.md).

Note: the wrapr right-dispatch we are using is only triggered when the right-hand side of a pipeline is a symbol or name. This is consistent with pipelines such as "5 %.>% sin" where we are not so much piping into the sin-function, but into a name that refers to the sin-function. However, piping into names covers most practical cases.

We can apply processing pipelines piece by piece.

pred\_step <- pkgfn(

"stats::predict.lm",

arg\_name = "newdata",

args = list(object = model))

exp\_step <- pkgfn(

"base::exp",

arg\_name = "x")

d2 %.>% pred\_step %.>% exp\_step

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

We can also build such a pipeline by piping pieces into each other.

model\_p <- pred\_step %.>% exp\_step

cat(format(model\_p))

## UnaryFnList(

## stats::predict.lm(newdata=., object),

## base::exp(x=., ))

d2 %.>% model\_p

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

The pipe notation is not required (but is a nice notation). The apply a list of function objects effect can be achieved directly with wrapr::ApplyTo().

ApplyTo(model\_p, d2)

## 1 2 3 4 5

## 18.63960 51.31337 141.26173 388.88260 1070.56368

The idea is: the processing pipelines store an arbitrary number of function objects as a simple list. The list declares function-like behavior to both ApplyTo and the wrapr dot-arrow pipe through R S3/S4 class declarations. Function objects do not capture environments as function closures do (though obviously any function in them does have its own closure). List of function objects can be easier to work with, store, and share than function closures or other pipeline structures.

We can look at the contents of a pipeline as follows.

model\_p@items

## [[1]]

## [1] "stats::predict.lm(newdata=., object)"

##

## [[2]]

## [1] "base::exp(x=., )"

In addition to the PartialNamedFn class we suggest looking at the following additional adapters:

* srcfn() which accepts the source code for an arbitrary expression (quoted either with quote-marks or with wrapr::qe()).
* wrapfn() class which directly accepts a function (including the closure).

Examples include:

s4 <- srcfn(

qe(. + y),

arg\_name = ".",

args= list(y=13))

print(s4)

## [1] "SrcFunction{ . + y }(.=., y)"

22 %.>% s4

## [1] 35

s5 <- wrapfn(

tan,

arg\_name = "x")

print(s5)

## [1] "PartialFunction{tan}(x=., )"

1:4 %.>% s5

## [1] 1.5574077 -2.1850399 -0.1425465 1.1578213

For convenience wrapr [dot-pipe](https://winvector.github.io/wrapr/reference/dot_arrow.html) pipeable object can be converted into a single-argument function of "dot" with the [as\_fn()](https://winvector.github.io/wrapr/reference/as_fn.html) method:

f5 <- as\_fn(s5)

f5(1:5)

## [1] 1.5574077 -2.1850399 -0.1425465 1.1578213 -3.3805150

1:5 %.>% f5

## [1] 1.5574077 -2.1850399 -0.1425465 1.1578213 -3.3805150

The idea is wrapr supplies many possible variations of notations: functions sequences as lists, function composition by pipe, function composition by call, function application by pipe, and function application by call. Then the user can pick what notation they prefer. rquery pipelines are very restricted (they date data.frames to data.frames and pre-check a number of invariants). UnaryFn pipelines are more free-form, they check very little before application.

The demonstrated design and functionality is inspired by partially applied functions, but a bit more circumspect in what is carried around. In Lisp "code is data", in R it is a bit more complicated- so a pure-data solution has some merits.

And these are the basics of wrapr function objects.