Practical 2

Jumping Rivers

Advanced R programming: Practical 2

Argument matching

R allows a variety of ways to match function arguments. For example, by position, by complete name, or by partial name. We didn't cover argument matching in the lecture, so let's try and figure out the rules from the examples below. First we'll create a little function to help

```
arg_explore = function(arg1, rg2, rg3)
    paste("a1, a2, a3 = ", arg1, rg2, rg3)
```

Next we'll create a few examples. Try and predict what's going to happen before calling the functions. One of these examples will raise an error - why?

```
arg_explore(1, 2, 3)
arg_explore(2, 3, arg1 = 1)
arg_explore(2, 3, a = 1)
arg_explore(1, 3, rg = 1)
```

Can you write down a set of rules that R uses when matching arguments?

```
## SOLUTION
## See http://goo.gl/NKsved for the offical document
## To summeriase, matching happens in a three stage pass:
#1. Exact matching on tags
#2. Partial matching on tags.
#3. Positional matching
```

Following on from the above example, can you predict what will happen with

```
plot(type = "l", 1:10, 11:20)
```

and

```
rnorm(mean = 4, 4, n = 5)
```

```
## SOLUTION
#plot(type="l", 1:10, 11:20) is equivilent to
plot(x=1:10, y=11:20, type="l")
#rnorm(mean=4, 4, n=5) is equivilent to
rnorm(n=5, mean=4, sd=4)
```

Functions as first class objects

Suppose we have a function that performs a statistical analysis

```
## Use regression as an example
stat_ana = function(x, y) {
    lm(y ~ x)
}
```

However, we want to alter the input data set using different transformations. For example, the log transformation. In particular, we want the ability to pass arbitrary transformation functions to `stat_ana`.

- Add an argument `trans` to the `stat_ana()` function. This argument should have a default value of `NULL`.

```
## SOLUTION
stat_ana = function(x, y, trans=NULL) {
  lm(y ~ x)
}
```

- Use `is.function()` to test whether a function has been passed to `trans`, transform the vectors `x` and `y` when appropriate. For example,

```
stat_ana(x, y, trans = log)
```

would take log's of `x` and `y`.

```
## SOLUTION
stat_ana = function(x, y, trans=NULL) {
  if(is.function(trans)) {
    x = trans(x)
    y = trans(y)
  }
  lm(y ~ x)
}
```

- Allow the `trans` argument to take character arguments in additional to function arguments. For example, if we used `trans = 'normalise'`, then we would normalise the data i.e. subtract the mean and divide by the standard deviation.

```
## SOLUTION
stat_ana = function(x, y, trans=NULL) {
  if(is.function(trans)) {
    x = trans(x)
    y = trans(y)
```

```
  } else if (trans == "normalise") {
    x = scale(x)
    y = scale(y)
  }
  lm(y ~ x)
}
```

Variable scope

Scoping can get tricky. **Before** running the example code below, predict what is going to happen

1. A simple one to get started

```
f = function(x) return(x + 1)
f(10)
```

```
##Nothing strange here. We just get
f(10)
```

1. A bit more tricky

```
f = function(x) {
    f = function(x) {
        x + 1
    }
    x = x + 1
    return(f(x))
}
f(10)
```

1. More complex

```
f = function(x) {
    f = function(x) {
        f = function(x) {
            x + 1
        }
        x = x + 1
        return(f(x))
    }
    x = x + 1
    return(f(x))
}
f(10)
```

```
## Solution: The easiest way to understand is
## to use print statements
```

```
f = function(x) {
    f = function(x) {
        f = function(x) {
            message("f1: = ", x)
            x + 1
        }
        message("f2: = ", x)
        x = x + 1
        return(f(x))
    }
    message("f3: = ", x)
    x = x + 1
    return(f(x))
}
f(10)
```

```
## f3: = 10

## f2: = 11

## f1: = 12
```

1.

```
f = function(x) {
    f = function(x) {
        x = 100
        f = function(x) {
            x + 1
        }
        x = x + 1
        return(f(x))
    }
    x = x + 1
    return(f(x))
}
f(10)
```

```
## Solution: The easiest way to understand is
## to use print statements as above
```

Function closures

Following the examples in the notes, where we created a function closure for the normal and uniform distributions. Create a similar closure for

- the Poisson distribution,[1]

```
poisson = function(lambda) {
    r = function(n = 1) rpois(n, lambda)
    d = function(x, log = FALSE) dpois(x, lambda,
        log = log)
    return(list(r = r, d = d))
}
```

[1] Hint: see ‘rpois‘ and ‘dpois‘.

- and the Geometric distribution.[2]

```
geometric = function(prob) {
    r = function(n = 1) rgeom(n, prob)
    d = function(x, log = FALSE) dgeom(x, prob,
        log = log)
    return(list(r = r, d = d))
}
```

[2] Hint: see ‘rgeom‘ and ‘dgeom‘.

Multiple column types

In the below code, I've attempted to loop through a data frame and extract the maximum values.

```
dd = data.frame(w = rnorm(10), x = letters[1:10],
    y = rnorm(10), z = rnorm(10))

max_cols = rep(NA, ncol(dd))
for (i in seq_along(dd)) {
    max_cols[i] = max(dd[, i])
}
max_cols
```

However, there's something wrong. The second column isn't numeric and so the for loop breaks when we get to there. Of course, we could just change the iterations to c(1,3,4) to leave out the second column. But imagine we have tens of columns. Use the `try()` function to bypass the error.

```
dd = data.frame(w = rnorm(10), x = letters[1:10],
    y = rnorm(10), z = rnorm(10))

max_cols = rep(NA, ncol(dd))
for (i in seq_along(dd)) {
    try(max_cols[i] <- max(dd[, i]))
}
max_cols
```

Solutions

Solutions are contained within the course package

```
library("jrAdvPackage")
vignette("solutions2", package = "jrAdvPackage")
```