quickcheck Package

Craig Parman

2/24/2015

quickcheck **Package**

Introduction

The package quickcheck from [RevolutionAnalytics] http://www.revolutionanalytics.com adds a twist to unit testing. Rather than specifing exact values to be tested, set of random values of a particular type is passed.

Here is a simple example. Let's test thet the funtion returns a numeric value. A simple test using testthat might look like:

```
library(testthat)

test_that("rest for floor", expect_identical(is.numeric(floor(3.14)), TRUE))
```

Here we specify the exact value 3.14 to be passed to the function floor. With quickcheck you don't specify the value, but rather the type of a set of random variables to be passed to the function. The function, `test`, uses randomly generate input values created by a generator function to test the assertion and collects the result.

```
library(quickcheck)

test(function(x = rdouble())  identical(is.numeric(floor(x)), TRUE))
```

```
## Pass  function (x = rdouble())
##  identical(is.numeric(floor(x)), TRUE)

## [1] TRUE
```

The test is not being run on a single value for x, but a series of values of type double generated by rdouble.

In the example, the generator `rdouble` generated a list of random value of type double and passes then to `identical(is.numeric(floor(x)), TRUE)`. The these componts together form the assertion. The assertion must return a logical vector of length 1. Now tha you get the idea lets look at the components closer.

Building tests

The main function in quickcheck in test. The usage is:

'test(assertion, sample.size = 10, stop = !interactive())'.

The `assertion` is a function and must return a single logical value for each time the assertion is tested. The returned logical value determines if the test has passed. The value for `sample.size` determines how many time to check the assertion and `stop` determines where a failed test and the associated error will stop R execution.

When the assertion returns TRUE value returned by `test` is `TRUE`. If the assertion returns `FALSE` the output depends on the `stop` parameter. If `stop` is `FALSE` a list with three elements is returned consisting of 1) the assertion 2) a list of in scope variabels (not fully implemented as of Feb. 2015), 3) List of arguments for each run. If the stop is `TRUE` a file is created which contants the list with the above values and is named `test.cases`. This file can be used to replicate errors. The error messsage generated reports the file names.

The most trival test that we could contruct would be:

```r
test(function()TRUE)
```

```
## Pass  function ()
##  TRUE
```

```
## [1] TRUE
```

Here we are just returning TRUE, so the test is passed. In fact we ran the test 10 times, as that is the default for how many time the assertion is checked. If we wan to so see that we can run.

```r
t<-test(function()FALSE, sample.size = 2, stop=FALSE)
```

```
## FAIL: assertion:
## function ()
## FALSEFAIL: assertion:
## function ()
## FALSE
```

In the output see that the test failed twice and there are 2 cases in the cases portion of the output list.

```r
t<-test(function(x=rdouble())FALSE, sample.size = 2, stop=FALSE)
```

```
## FAIL: assertion:
## function (x = rdouble())
## FALSEFAIL: assertion:
## function (x = rdouble())
## FALSE
```

The generator, in this case `rdouble` can take on many different forms and the package provide serveral built in and a mechanism to create your own.

In fact you not only get a random set of doubles, you get a randon number of values. To see this lets run rdouble twice. We will set the random number seed to 0 for reproducibility.

```r
library(quickcheck)

set.seed(0)

rdouble()
```

```
##  [1] -0.326233361  1.329799263  1.272429321  0.414641434 -1.539950042
##  [6] -0.928567035 -0.294720447 -0.005767173  2.404653389  0.763593461
## [11] -0.799009249 -1.147657009 -0.289461574
```

```r
rdouble()
```

```
## [1] -0.4115108  0.2522234 -0.8919211  0.4356833 -1.2375384 -0.2242679
## [7]  0.3773956  0.1333364  0.8041895
```

In the first case we get 13 values and in the second we get 9. We can further control the number generation with the element and size parameters. The element value sets expectation of the values and size sets the average number of values returned as displayed below.

```r
library(quickcheck)

set.seed(0)

set1 <- rdouble(element=100,size=10)

set2 <-rdouble(element=10,size=100)

mean(set1)
```

```
## [1] 100.0657
```

```r
length(set1)
```

```
## [1] 13
```

```r
mean(set2)
```

```
## [1] 10.12117
```

```r
length(set2)
```

```
## [1] 97
```

We will look at the generators more depply in the section in generators.

Usage

Defining Generators

Generator ceate the values that are tested against. There are built in generators for most data types. If needed custom generators can be generated for cases where specific values, or mixed types are needed. On thing to keep in min is that these generators not only return randome values, but also random numbers of values. This can be overridden, but the default behavior is to return an random number of random variable as defined by the call.

Built in genertators Below is the list of buitl in generators. There are atomic, 2 dimensional, and lists avaialble. Then a type can consructed from a variaty of atomic types, each call to that generator will create a

Generator	Elements	Usage
rany	all generators	rany() , rany(list(rdouble,rinteger))
rcharacter	characters	rcharacter(element = 10, size = 10)
rfactor	factors	rfactor(element = 10, size = 10)
constant	constant value	constant(const = 3)()
rdata.frame	data.frame	rdata.frame(element = ratomic, nrow = 10, ncol = 5)
rmatrix	matrix of atomics	rmatrix(element, nrow, ncol)
ratomic	atomics	ratomic(element = atomic.generators, size = 10)
rdouble	doubles	rdouble(element = 0, size = 10)
rDate	Dates	rDates(element = list(from=Sys.Date()-10,to=Sys.Date())
rinteger	integers	rinteger(element = 100, size = 10)
rnumeric	all numerics	rnumeric(element = 100, size = 10)
rlist	list of atomics	rlist(), rlist(element=list(rdouble(siz=2)))
rlogical	logicals	rlogical(element = 0.5, size = 10)
rraw	raw values	rraw(element = as.raw(0:255), size = 10)

You have the ability to fine tune the generators with the size and element parameters. The element parameter can be used to control the values goint into the list. The values for element can be a single value, a vector, or a random number generator.

```
rinteger(element = 30:50,size =10 )
```

```
##  [1] 27 36 25 31 36 33 32 30 36 38 44 40 41 38
```

```
rdouble(element = rnorm(10,mean=2,sd=.2),siz=20 )
```

```
##  [1] 2.3708460 1.5811096 1.0730802 0.7892705 1.3700844 0.6891121 1.3054844
##  [8] 2.2803580 0.6455527 3.2881209 0.9909139 1.8083354 4.5293032 2.6958260
## [15] 1.8856914 0.9430684 2.3362446 1.7960923 1.4511887 2.1357496 1.8135288
## [22] 4.5635184 3.8398615
```

There are two helper function for generators. The first is `mixture`. 'Mixture' creates a random generator that creates an equally weighted mixture distribution of the generators. Repeated calls will randomly generate vaules of one one the generators. you can also use a formula syntax to pass parameters to the generating function as shown in the last example. This allows us to ensure that exactly size number of values are returned otherwise recycling is used.

```
set.seed(0)

mixture(list(rlogical,rcharacter))()
```

```
## [1] "58ef5fa"            "3ff2f39aa"          "b5c4a2dc3"
## [4] "ef60e556939ee3d7c" "09100442c5d8"      "072466c"
## [7] "d62f7a93"           "7a"
```

```
mixture(list(rlogical,rcharacter))()
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

```
#pass the size parameter to runif

rdouble(element = ~runif(size, min = -1))
```

```
##  [1]  0.57871246 -0.95333760 -0.04553987  0.46462748  0.38546311
##  [6] -0.04476076  0.72241895 -0.12380579 -0.51040545 -0.85864191
## [11] -0.80106768
```

The `size` parameter can be used to control the expectation for the number of elements returned. Ror example 'rdouble(size = 100)' would eturn vectors of doubles with an aveage length of 100. You can get an exact number of values by setting the size parameter with the constant function of the functional notation ~.

```
rdouble(size = constant(4))
```

```
## [1] -0.4781501  0.4179416  1.3586796 -0.1027877
```

```
rdouble(size = ~4)
```

```
## [1]  0.38767161 -0.05380504 -1.37705956 -0.41499456
```

For nested objects such as list you can add the height parameter. This control the maximun depth of nesting. For 2 dimentional obljects size is replaced with nrow and ncol, both with the same behavior as size.

```
set.seed(0)

rlist(height=4,size=2)
```

```
## [[1]]
##  [1] -0.8356286  1.5952808  0.3295078 -0.8204684  0.4874291  0.7383247
##  [7]  0.5757814 -0.3053884  1.5117812  0.3898432
##
## [[2]]
## [1] -0.2992151 -0.4115108  0.2522234 -0.8919211  0.4356833 -1.2375384
## [7] -0.2242679  0.3773956  0.1333364
##
## [[3]]
## [1] 75ae4800b  a5b11      cd1a8467ab
## Levels: 75ae4800b a5b11 cd1a8467ab
##
## [[4]]
##  [1] "552307904eb8c" "78c5fec09d6d"  "a24594d2e"     "19675c91a"
##  [5] "12628fdf0740"  "cc58f52bf7d"   "dd41241"       "96654"
##  [9] "bddc9e445a45"  "d8f5dc9f1"     "3469c6beb817"  "a7d3c9776ba"
## [13] "f4cbe8af"      "629eb8e645d"
```

```
rmatrix(nrow=~2,ncol=~4)
```

```
##      [,1]           [,2]           [,3]      [,4]
## [1,] "090b99892f14" "346868d23e00" "43c6cac" "5b769dde7f4ef4"
## [2,] "3c84a0330"    "2c7b19ec50"   "a6396"   "e2eb07c176"
```

The function allows you to randomly select vaules from a vector of values with or without replacement.

```
select(c(1,TRUE,"gg",4),replace=TRUE)(size=10)
```

```
##  [1] "4"    "gg"   "gg"   "gg"   "1"    "TRUE" "gg"   "TRUE" "1"    "gg"
```

Constructing custom generators The built in generator can be used to contruct a new generator. For examples we can combine use mixture to create a list with integers and characters.

```
set.seed(0)
rlist(mixture(list(rinteger,rcharacter)),size=constant(3))
```

```
## [[1]]
## [1] "58ef5fa"           "3ff2f39aa"         "b5c4a2dc3"
## [4] "ef60e556939ee3d7c" "09100442c5d8"      "072466c"
## [7] "d62f7a93"          "7a"
##
## [[2]]
## [1]  99 109 108 105 109 107 100  80 105
##
## [[3]]
## [1] "1a9503e4"        "1e45d1d"         "89c3f"           "ce3abbc800508"
## [5] "c8845994"        "751165d47f9e6"   "416360f03"       "672422e2514f235"
## [9] "552307904eb"
```

Let say we need a generator that constructs a matrix of integers. We can use the rinteger generator to create the values for the matrix. We must fix the length of the values returned by rinteger to nrow*ncol so we don't recycle values. We will call the new function rIntMatrix

```
rIntMatrix<-function(element=100,nrow=5,ncol=5)
{
  nr<-quickcheck:::rsize(nrow)
  nc<-quickcheck:::rsize(ncol)

 matrix( rinteger(element=element, size=constant(nc*nr)) ,nr,nc)

}

set.seed(0)

rIntMatrix(element=50,nrow=4,ncol=5)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   47   47   55   57
## [2,]   44   44   54   45
## [3,]   52   45   56   46
## [4,]   39   47   55   48
## [5,]   53   49   50   46
## [6,]   55   49   35   52
## [7,]   54   56   53   49
```

Defining Assertions

The assertion you create with quickcheck can leverage the expectations from the testthat package. Quickcheck used a set derived from the testthat package. They use the same name exect thet the leading expect_ is removed.

Expectation Types in Quickcheck equal equal_to_reference equivalent error false identical is less_than match message more_than named null output that true warning

The use of the test is pretty obvious. Details for the specific cases can be found in the testthat help. Below are a few simple tests that use different expectations that highlight the usage.

```r
test(function(x=rcharacter()) expect("error", 1/x ))
```

```
## Pass  function (x = rcharacter())
##  expect("error", 1/x)
```

```
## [1] TRUE
```

```r
test(function(x=rcharacter()) expect("named", data.frame(data=x) ))
```

```
## Pass  function (x = rcharacter())
##  expect("named", data.frame(data = x))
```

```
## [1] TRUE
```

```r
test(function(x=rcharacter()) expect("match", x,"[:alnum:]*" ))
```

```
## Pass  function (x = rcharacter())
##  expect("match", x, "[:alnum:]*")
```

```
## [1] TRUE
```

```r
 test(function(x=rinteger()) expect("less_than",sum(x),sum(x+1)))
```

```
## Pass  function (x = rinteger())
##  expect("less_than", sum(x), sum(x + 1))
```

```
## [1] TRUE
```

Debugging Failed Tests

Unfortunately your code will most likely not pass every test. when a test fails quickcheck lets you know what failed and which specfic cases failed. Here is a test that is sure to fail.

```r
set.seed(0)
test(function(x=rinteger()) expect("more_than",sum(x),sum(x+1)), stop = FALSE, sample.size=2)
```

```
## FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))

## $assertion
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))
##
## $env
## NULL
##
## $cases
## $cases[[1]]
## $cases[[1]]$x
##  [1]  96  91 104  84 104 107 105  96  92  93  97  99  99
##
##
## $cases[[2]]
## $cases[[2]]$x
##  [1] 108 105 109 107 100  80 105 110  93  95  97  98
```

We as expected the test failed and the test function reported which tests failed and the value for each case. We can caputure this outout for later debuging. Here the output is captured in out. The function repro loads this case and starts he debugger.

```r
set.seed(0)
out = test(function(x=rinteger()) expect("more_than",sum(x),sum(x+1)), stop = TRUE ,sample.size=2)
```

```
## FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))

## Error in test(function(x = rinteger()) expect("more_than", sum(x), sum(x + : load("/home/craig/Docume
```

```r
repro(out)
```

```
## Error in repro(out): object 'out' not found
```

```r
set.seed(0)
test(function(x=rinteger()) expect("more_than",sum(x),sum(x+1)), stop = TRUE,sample.size=2)
```

```
## FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))FAIL: assertion:
## function (x = rinteger())
## expect("more_than", sum(x), sum(x + 1))
## Error in test(function(x = rinteger()) expect("more_than", sum(x), sum(x + : load("/home/craig/Docum
```

With stop set to true we get a error message and a invetation to load a file that was automatically created. The file containt the cases from the failed tests in a variable called test.cases that can loaded with repro.