biggr

Freddy Drennan

2019-05-05

Contents

Chapter 1

About the biggr Package

My vision for this package has somewhat grown from what I intended to what it is today. At first, my intention was just to make some convenience functions for dealing with AWS, but what I soon realized is that there is a lot of potential for more sophisticated tools once these pieces are in place. For example, you can spin up an EC2 server with the `ec2_instance_create` function. That function has a useful parameter in it, `user_data`. With that parameter, you can send a shell script to run while the server is starting up. Most will go ahead and log into the server, install packages, databases, Rstudio Server, GPU Support, PostrgreSQL, etc. This is not always intuitive or easy. Especially for the beginner.

My hope is I can take this burden from the day-to-day data practitioner. Most people know what they want - access to computing power - but aren't familiar with all the technical pieces of the puzzle.

1.1 WARNING

This package uses AWS which is not a free service. You assume all costs associated and should do your research about what you are requesting. I'm not picking up the bill.

Chapter 2

Installation

Before we can do anything, we need to install the package. The package is currently located on my github page. If you have it installed `devtools`, please run install.packages("devtools").

```
devtools::install_github('fdrennan/biggr')
```

```
## Skipping install of 'biggr' from a github remote, the SHA1 (4f34d443) has not changed since last ins
##   Use `force = TRUE` to force installation
```

Once this is complete, we will use two packages. `biggr` and `reticulate`. For more on reticulate, visit Rstudio.

```
library(biggr)
library(reticulate)
```

Next use the `install_python` and choose an environment name to install the package. By default, R uses the virtual environment `r_reticulate`. You can use this environment or specify a different name. I will use a different name below. My understanding is that if you do not specify the environment, each time you load up R you will not have to specify which environment to use because R uses `r_reticulate` by default. *You can also use Conda by specifying `method = conda`. The insides of the function are directly below.

```
install_python
```

```
## function (method = "auto", conda = "auto", envname = "r_reticulate")
## {
##     reticulate::py_install("scipy", method = method, conda = conda,
##         envname = envname)
##     reticulate::py_install("awscli", method = method, conda = conda,
##         envname = envname)
##     reticulate::py_install("boto3", method = method, conda = conda,
##         envname = envname)
## }
## <bytecode: 0x7fce2c638150>
## <environment: namespace:biggr>
```

Here we create a virtual environment named `biggr`.

```
install_python(envname = 'biggr')
```

Next we go ahead and set which environment we want to use.

```
use_virtualenv('biggr')
```

Here we verify that we are in fact using the correct environment.

```
py_config()
```

```
## python:         /Users/digitalfirstmedia/.virtualenvs/biggr/bin/python
## libpython:      /System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/config/libpyt
## pythonhome:     /System/Library/Frameworks/Python.framework/Versions/2.7:/System/Library/Frameworks/
## virtualenv:     /Users/digitalfirstmedia/.virtualenvs/biggr/bin/activate_this.py
## version:        2.7.10 (default, Aug 17 2018, 19:45:58)  [GCC 4.2.1 Compatible Apple LLVM 10.0.0 (cl
## numpy:          /Users/digitalfirstmedia/.virtualenvs/biggr/lib/python2.7/site-packages/numpy
## numpy_version:  1.16.3
## scipy:          /Users/digitalfirstmedia/.virtualenvs/biggr/lib/python2.7/site-packages/scipy
##
## python versions found:
##  /Users/digitalfirstmedia/.virtualenvs/biggr/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/r-reticulate/bin/python
##  /usr/bin/python
##  /usr/local/bin/python
##  /usr/local/bin/python3
##  /Users/digitalfirstmedia/.virtualenvs/biggr_env/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/py3-virtualenv/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/r_reticulate/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/r-tensorflow/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/venv/bin/python
##  /Users/digitalfirstmedia/.virtualenvs/voicecode/bin/python
##  /anaconda3/envs/r-tensorflow/bin/python
##  /Users/digitalfirstmedia/.talon/bin/python
```

2.1 Authentication with AWS

Next we set permissions. You need to go to the AWS console and create a user which has administrator access. This will set the environment variables for the underlying aws command line interface.

```
configure_aws(aws_access_key_id = "XXXXXXXXXX",
              aws_secret_access_key = "XXXXXXXXXX",
              default.region = "us-east-2")
```

Then finally we verify that the package is in fact working in talking to Python as well as AWS.

```
s3_list_buckets()
```

```
## # A tibble: 4 x 2
##   name              creation_date
##   <chr>             <chr>
## 1 couch-dog-photos 2019-03-08 04:45:05+00:00
## 2 fdrennanunittest 2019-05-02 21:15:04+00:00
## 3 freddydbucket    2019-05-03 05:46:19+00:00
## 4 kerasmods        2019-01-29 20:47:11+00:00
```

Chapter 3

boto3 and Python

This package is built off of `boto` and `reticulate`. Once you have successfully installed `biggr`, you immediately have complete access to AWS, though in a somewhat complicated form. Let's send ourselves a text message.

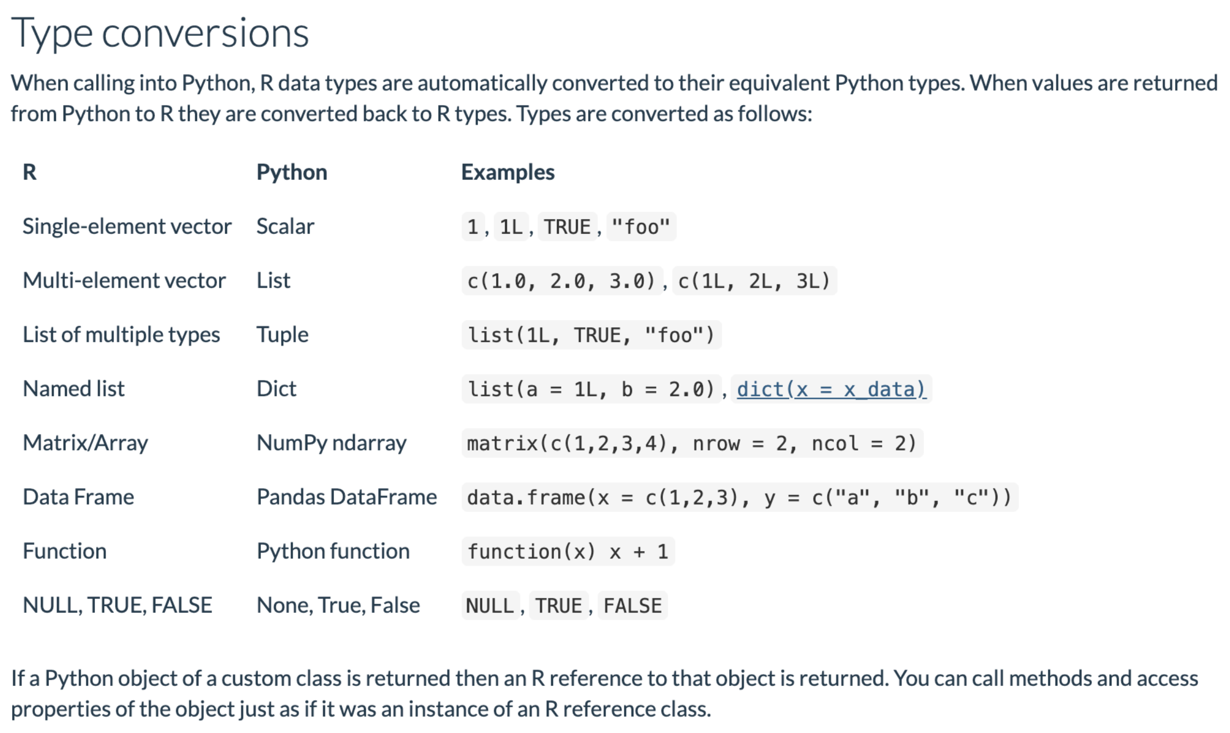
```r
library(biggr)
library(reticulate)
library(tidyverse)
```

Use the `boto3` function to gain access to the underlying Python module.

```r
boto = boto3()
```

Following the tutorial located here, we see how incredibly powerful this package could be.

More or less, to write Python in R all you need to do is replace the .'s with $'s However it's good to know type conversion between the two languages. This is taken from the reticulate documentation.



Type conversions

When calling into Python, R data types are automatically converted to their equivalent Python types. When values are returned from Python to R they are converted back to R types. Types are converted as follows:

R	Python	Examples
Single-element vector	Scalar	`1`, `1L`, `TRUE`, `"foo"`
Multi-element vector	List	`c(1.0, 2.0, 3.0)`, `c(1L, 2L, 3L)`
List of multiple types	Tuple	`list(1L, TRUE, "foo")`
Named list	Dict	`list(a = 1L, b = 2.0)`, `dict(x = x_data)`
Matrix/Array	NumPy ndarray	`matrix(c(1,2,3,4), nrow = 2, ncol = 2)`
Data Frame	Pandas DataFrame	`data.frame(x = c(1,2,3), y = c("a", "b", "c"))`
Function	Python function	`function(x) x + 1`
NULL, TRUE, FALSE	None, True, False	`NULL`, `TRUE`, `FALSE`

If a Python object of a custom class is returned then an R reference to that object is returned. You can call methods and access properties of the object just as if it was an instance of an R reference class.

Client is the lowest level within the boto module, read more about it here for sns.

```
client <- boto$client('sns', region_name='us-east-1')
```

```
response <- client$publish(
    PhoneNumber="+15555555555",
    Message="Hello World!!!!"
)
```

The AWS responses are pretty dirty. This is one of the issues I'm working on. We can take the response and clean it up for others.

```
response
```

```
## $ResponseMetadata
## $ResponseMetadata$RetryAttempts
## [1] 0
##
## $ResponseMetadata$HTTPStatusCode
## [1] 200
##
## $ResponseMetadata$RequestId
## [1] "2ab6b37b-21d9-5802-b257-e63d9900353f"
##
## $ResponseMetadata$HTTPHeaders
## $ResponseMetadata$HTTPHeaders$`x-amzn-requestid`
## [1] "2ab6b37b-21d9-5802-b257-e63d9900353f"
##
## $ResponseMetadata$HTTPHeaders$date
## [1] "Fri, 03 May 2019 21:46:01 GMT"
##
## $ResponseMetadata$HTTPHeaders$`content-length`
## [1] "294"
##
## $ResponseMetadata$HTTPHeaders$`content-type`
## [1] "text/xml"
##
##
##
## $MessageId
## [1] "68561ace-e6fc-50e8-8b15-4f885e258dd5"
```

While this is simplified, an R user will find this much more intuitive.

```
send_sms <- function(phone_number = NA,
                     message = "Hello World!",
                     region = "us-east-1",
                     message_aws = FALSE) {
  client <- boto$client('sns', region_name=region)
  phone_number <- paste0("+", phone_number)
  response <- client$publish(
      PhoneNumber = phone_number,
      Message     = message
  )
  if(message_aws) {
    return(response)
  } else {
    return(TRUE)
  }
}
```

Doesn't this just seem a lot better? Wrapping up the most useful parts of Python code into R is the first step.

```
send_sms(phone_number = 15555555555, message = "Hi, how are you?")
```

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