reticulate allows us to toggle between R and python in the same session, callling R objects when running python scripts and vice versa. When calling R data structures in python, the R structures are converted to the equivalent python structures where applicable. However, like translating English to Mandarin, translating R structures to python may not be straightforward which we will see later.

There are 5 R data structures:

1. vector (more specifically atomic vector)
2. list
3. array
4. matrix (special kind of array which is 2 dimensional)
5. data frame

In this post, we will look at translating R’s vector into python.

# load libraries

library(tidyverse)

library(reticulate)

**A R vector is a python …**

well it depends if the R vector has single or multiple elements.

**Single element R vector**

If the R vector has only 1 element, the python structure will be a scalar. A scalar is a structure which contains a single value. The value can be any type e.g. 69, 0.07, or ‘banana’.

Let’s verify with some code. Is Rvec\_1 an atomic vector?

Rvec\_1<-1

is.vector(Rvec\_1)

## [1] TRUE

is.atomic(Rvec\_1)

## [1] TRUE

Indirectly, you can print the class() of the object. If it prints the element type, you can infer the object is a vector.

class(Rvec\_1)

## [1] "numeric"

Is a single element R vector a python scalar structure?

py\_run\_string("import numpy as np")

py\_eval("np.isscalar(r.Rvec\_1)")

## [1] TRUE

Likewise, you can print the type of the object. If it prints the element type, you can infer the structure is a scalar.

py\_eval("type(r.Rvec\_1)")

##

If you wish to run everything in R and achieve the above, you will have to convert the R object into a python object and store this converted object in your R’s global environment.

r\_to\_py(Rvec\_1) %>% class()

## [1] "python.builtin.float" "python.builtin.object"

There you have it. When you convert a single element R vector into python, it is a float element type which is indicative that it is a python scalar structure.

**Multi element R vector**

If the R vector has multiple elements, the python structure will be a list. Let’s assert this with some code. Is Rvec\_multi a R atomic vector? The class() is an element type thus it can be inferred to be a R vector.

Rvec\_multi<-c(66,99, 0.07)

class(Rvec\_multi)

## [1] "numeric"

Is a multi element R vector a python list? Yes, it is.

r\_to\_py(Rvec\_multi) %>% class()

## [1] "python.builtin.list" "python.builtin.object"

**Named vectors**

Occasionally, you may work with named vectors in R; for instance, when calculating quantiles.

(Rvec\_name<-quantile(rnorm(100)))

## 0% 25% 50% 75% 100%

## -3.20985677 -0.60767955 0.03261286 0.77174145 2.58177907

Named vectors are still considered vectors.

Rvec\_name %>% class()

## [1] "numeric"

Do note that the names in the named vectors (e.g. 0%, 25%..) are treated as character and NOT numbers.

Rvec\_name %>% str()

## Named num [1:5] -3.2099 -0.6077 0.0326 0.7717 2.5818

## - attr(\*, "names")= chr [1:5] "0%" "25%" "50%" "75%" ...

However, python ignores the names when translating a multi element named vector. python treats it like another python list.

r\_to\_py(Rvec\_name)

## [-3.2098567735297747, -0.6076795490223229, 0.032612857512085064, 0.7717414498796162, 2.5817790740464406]

**Some differences between python and R**

**Element types**

We have been using element types to infer if the object is a R vector or a python scalar. Thus, it would helpful to know some of the differences between R and python element types.

**Element types (numbers)**

By default, R treats numbers as floats/numerics regardless if they are whole numbers or numbers with decimals

class(1)

## [1] "numeric"

class(0.07)

## [1] "numeric"

On the other hand, python treats whole numbers as integers.

py\_eval("type(1)")

##

Python treats number with decimals just like R, as floats/numerics

py\_eval("type(0.07)")

##

The trick for R to treat whole numbers as integers in the eyes of both R and python is to add the suffix L after the number.

Rvec\_1int<-1L

class(Rvec\_1int)

## [1] "integer"

r\_to\_py(Rvec\_1int) %>% class()

## [1] "[python.builtin.int](http://python.builtin.int)" "python.builtin.object"

**Element types(coercing)**

Elements in multi element R vectors adhere to singularity. In other words, different element types are coerced such that all elements have the same type.

Let’s look at an example. First, I will create 3 single element vectors of different element types.

Relement\_int=2L

class(Relement\_int)

## [1] "integer"

Relement\_bool=TRUE

class(Relement\_bool)

## [1] "logical"

Relement\_char="banana"

class(Relement\_char)

## [1] "character"

Next, I will combine these vectors into a multi element vector. Let’s reassess the element type for each element.

Rvec\_mix<- c(Relement\_int, Relement\_bool, Relement\_char)

class(Rvec\_mix[1])

## [1] "character"

class(Rvec\_mix[2])

## [1] "character"

class(Rvec\_mix[3])

## [1] "character"

As you can see, all the different elements have been coerced into the same element type when they are combined in a multi element vector. Often, the individual elements are coerced into strings as strings is the most accommodating element type.

In contrast, python doesn’t coerce element types when lists are created. The integrity of each element type remains unchanged.

py\_run\_string("Plist\_mix=(r.Relement\_int, r.Relement\_bool, r.Relement\_char)")

py\_eval("type(Plist\_mix[0])")

##

py\_eval("type(Plist\_mix[1])")

##

py\_eval("type(Plist\_mix[2])")

##

**Indexing**

Besides the differences in element types, there are differences in indexing for each language.

**Indexing (zero/non-zero)**

R uses non-zero indexing

Rvec\_multi[1]

## [1] 66

python uses zero indexing

py\_eval("r.Rvec\_multi[0]")

## [1] 66

**Indexing (negative numbers)**

In addition to non-zero and zero indexing, there are other differences in indexing. In R, negative index number means that the element of that index number is excluded.

Rvec\_multi[-1]

## [1] 99.00 0.07

In python, negative index number means that indexing begins from the end of the dataset.

py\_eval("r.Rvec\_multi[-1]")

## [1] 0.07