5 - Subtle But Important Differences

R and Python have many differences but some of the major ones are surrounded around initializing and/or creating objects, editing/mutating objects and how those objects are stored in memory.

5.1 - Initializing Objects in R

R will initialize new memory for all created objects. The easiest way to show this is with a simple example initializing matrices. If we create a matrix, call it A and then call a new variable B and set it equal to A (`B <- A`), these matrices are actually "pointing" to different spaces in memory. Changing the contents of A will NOT change the contents of B. A and B will actually share the same space in memory until something is done to modify one of them, at which point it is copied to a new location in memory ("copy-on-modify" semantics).

```
A <- matrix(c(1,1,1,1), ncol = 2)
B <- A
print(A)
print(B)
```

```
     [,1] [,2]
[1,]    1    1
[2,]    1    1
     [,1] [,2]
[1,]    1    1
[2,]    1    1
```

```
A[1,1] <- 2 # only A is changed
print(A)
print(B)
```

```
     [,1] [,2]
[1,]    2    1
[2,]    1    1
     [,1] [,2]
[1,]    1    1
[2,]    1    1
```

Notice how only the contents of the matrix A are changed. R's built-in copy-on-modify process prevents users from having two symbols always pointing to the same object, the concept of pointers does not fit naturally into R's language concept. In order to achieve similar functionality to Python one needs the help of an external package 'pointr'.

5.1.1 R pointr Package

As per the pointr documentation page [12] pointr: Working Comfortably with Pointers and Shortcuts to R Objects "R has no built-in pointer functionality. The pointr package fills this gap and lets you create pointers to R objects, including subsets of data frames. This makes your R code more readable and maintainable." The pointr package provides functionality to create pointers to any R object easily, including pointers to subsets/selections from data frames.

Like other R packages, the pointr package can be installed in R or R Studio with the command `install.packages('pointr')` . Once the package is installed it can be loaded/imported in with the command `library(pointr)` after which all the package functions will be available for use. It should be noted that pointr package version 0.1.0 is being used.

```
# install.packages('pointr') -> run if not installed
library(pointr)
packageVersion("pointr")
```

```
[1] '0.1.0'
```

```
A <- matrix(c(1,1,1,1), ncol = 2)
ptr("A_ptr", "A") # A_ptr points to A
print(A)
print(A_ptr)
```

```
     [,1] [,2]
[1,]    1    1
[2,]    1    1
     [,1] [,2]
[1,]    1    1
[2,]    1    1
```

```
A[1,1] <- 15 # both change
print(A)
print(A_ptr)
```

```
     [,1] [,2]
[1,]   15    1
[2,]    1    1
     [,1] [,2]
[1,]   15    1
[2,]    1    1
```

```
A_ptr[1,1] <- 0 # both change
print(A)
print(A_ptr)
```

```
     [,1] [,2]
[1,]    0    1
[2,]    1    1
     [,1] [,2]
[1,]    0    1
[2,]    1    1
```

Notice how the contents of both matrices change when changing the values in one or the other.

5.2 - Initializing Objects in Python

Unlike R, Python does not initialize new memory for all created objects. The easiest way to show this is with a simple example initializing matrices. If we create a NumPy matrix, call it A and then call a new variable B and set it equal to A (`B = A`), both matrices are now "pointing" to the same space in memory. Changing the contents of A will also change the contents of B.

```
A = np.array([[1,1],[1,1]])
B = A
A, B
```

```
(array([[1, 1],
        [1, 1]]),
 array([[1, 1],
        [1, 1]]))
```

```
A[0,0] = 15 # both change
A,B
```

```
(array([[15,  1],
        [ 1,  1]]),
 array([[15,  1],
        [ 1,  1]]))
```

Notice how the contents of both matrices are changed. Performing this same sequence of code in R will produce two separate matrices in memory for A and B. Changing the contents of A would not affect B at all. This is a major difference between the two languages and can lead to coding bugs if this initialization is not done carefully. For Python to behave the same way as R in this case, one needs to make use of the NumPy `copy()` function. We wish to have the matrix B initialized with the same contents as A but be stored in separate memory as its own object.

```
A = np.array([[1,1],[1,1]])
B = np.copy(A)
A, B
```

```
(array([[1, 1],
        [1, 1]]),
 array([[1, 1],
        [1, 1]]))
```

```
A[0,0] = 15 # only A changes
A,B
```

```
(array([[15,  1],
        [ 1,  1]]),
 array([[1, 1],
        [1, 1]]))
```

Notice now only the contents of the matrix A are changed. It should be noted that the NumPy `copy()` function is a "shallow" copy, it will not copy object elements within array. In the example above we just have numerical arrays (matrix) which do not contain objects so the copy is performed correctly. If one element was added, say character 'a', then a `deepcopy()` all would be needed as the basic NumPy `copy()` no longer works.

```
A = np.array([[1,1],'a', [1,1]], dtype = object)
B = np.copy(A)
A, B
```

```
(array([list([1, 1]), 'a', list([1, 1])], dtype=object),
 array([list([1, 1]), 'a', list([1, 1])], dtype=object))
```

```
A[0][0] = 15 # changes both
A, B
```

```
(array([list([15, 1]), 'a', list([1, 1])], dtype=object),
 array([list([15, 1]), 'a', list([1, 1])], dtype=object))
```

Notice how the contents of both the matrix A and B are changed.

5.2.1 Python copy Module

Documentation for the copy module can be found at [7] where they provide more clarity of the difference between `copy()` and `deepcopy()` . "A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original."

```
import copy # object copying
```

```
A = np.array([[1,1],'a', [1,1]], dtype = object)
B = copy.deepcopy(A)
A, B
```

```
(array([list([1, 1]), 'a', list([1, 1])], dtype=object),
 array([list([1, 1]), 'a', list([1, 1])], dtype=object))
```

```
A[0][0] = 15 # only A changes
A, B
```

```
(array([list([15, 1]), 'a', list([1, 1])], dtype=object),
 array([list([1, 1]), 'a', list([1, 1])], dtype=object))
```

Notice how only the contents of the matrix A are changed.

5.3 - R Pre-Allocation vs. Appending

In general, when creating a list or an array of elements this can be done by appending to the front/back of the list or by updating specific elements. For example, a list of size three with all elements equal to zero ($[0, 0, 0]$) can be created by initializing an empty list ($[]$) and appending 0 to the front or back three times. Alternatively, this can be done by initializing a list of size three of any elements ($[e_1, e_2, e_3]$) then updating each element to 0. This section should be a major point of focus and interest to those concerned about the differences in computational run time for these two methods. In R, one should always pre-allocate space for an object and avoid appending whenever possible. An experiment in R is run below where the run time to create arrays of varying sizes is tracked. A method of iteratively appending values to an empty array is tested against a method of filling an array of a pre-specified size with the same values. Note: the arrays built in each iteration are just arrays of ones ([1,1,....,1]).

```
N <- c(2, 10, 100, 500, 1000, 5000, 10000, 20000, 50000, 100000)
app_time <- numeric(length(N))
pre_time <- numeric(length(N))
count <- 1
for (j in N) {
    l1 <- c()
    l2 <- numeric(j)

    t <- Sys.time()
    for (i in 1:j) {
        l1 <- append(l1, 1)
    app_time[count] <- Sys.time() - t
    }

    t2 = Sys.time()
    for (i in 1:j) {
        l2[i] = 1
    pre_time[count] <- Sys.time() - t2
    }

    count <- count + 1
}
```

A matrix: 9 x 3 of type dbl

Array of size N	Appending	Pre-Allocation
10	0.000236	0.000204
100	0.002469	0.002213
500	0.012527	0.011183
1000	0.027220	0.022216
5000	0.162233	0.114250
10000	0.399416	0.237125
20000	1.149267	0.443867
50000	5.199226	1.125084
100000	18.368219	2.252459

Note: computational run times will vary depending on the machine one uses but these overall trends should stay the same. One can notice the drastic difference in runtime when comparing these two methods. The difference in run time looks to increase as the size of the desired array increases. Clearly the pre-allocation method is superior to appending in R. This is because in R each time you append an element, R will create a new space in memory for the object, ie R will create a new object with size 1 greater than the old one and then copy each of the old object values into the new one.

5.4 - Python Pre-Allocation vs. Appending

Similar to what was done above in R, a small experiment is run in Python comparing the methods of pre-allocation and appending.

```
N = np.array([2, 10, 100, 500, 1000, 5000, 10000, 20000, 50000, 100000])
app_time = []
pre_time = []
for j in N:
    l1 =[]
    l2 = [0] * j

    t = time.time()
    for i in range(j):
        l1.append(1)
    t2 = time.time() - t
    app_time.append(t2)

    t3 = time.time()
    for i in range(j):
        l2[i] = 1
    t4 = time.time() - t3
    pre_time.append(t4)
```

Computational Runtime (seconds)

Array of size N	Appending	Pre-Allocation
10	0.000002	0.000001
100	0.000011	0.000009
500	0.000050	0.000043
1000	0.000102	0.000086
5000	0.000500	0.000455
10000	0.000999	0.001187
20000	0.002029	0.001843
50000	0.005026	0.004513
100000	0.009944	0.009273

Note: computational run times will vary depending on the machine one uses but these overall trends should stay the same. It is shown that pre-allocating in Python is only slightly faster then the appending approach but there is not much of a gain, the two methods are comparable. This is a completely different result compared to the experiment in R, which showed one should always pre-allocate. In Python this is something the user does not need to be as concerned about as the methods can be interchanged without significant change in run time.

[7] Van Rossum, G. & Drake, F.L., 2009. Python 3 Reference Manual, [link] (https://www.python.org/)
[12] Joachim Zuckarelli, 2020. pointr: Working Comfortably with Pointers and Shortcuts to R Objects, [link] (https://CRAN.R-project.org/package=pointr)