R vs Python: Recursive and Iterative Runtimes

"programmers often increase the conceptual complexity of a program in an effort to reduce its computational complexity" (John Guttag, *Introduction to Computation and Programming Using Python, 2nd edition*, p. 135).

Recursive Function Calls

Having been steeped in R prior to learning Python, and having already been awed by the speed of R for certain tasks--- even while being cognizant of how slow it can be moving through a for-loop or when retrieving from, or inserting into, a dataframe,---I was curious to compare R's runtime performance (version 3.5.3) with that of Python's (version 3.7.1) when handling recursive function calls.

Ideally the comparison should be made "all else being equal". For example, although both R and Python are object-oriented languages, most objects in R are not like those in Python (cf. p. 116 of Hadley Wickham's *Advanced R*). A typical R object such as a list is copied when passed to a function, whereas a list in Python is passed by reference---a difference that matters greatly when a function is called recursively. Because copying is computationally costly when the argument passed is quite large, there is reason to think Python will generally outperform R for many recursive tasks. Thus, a fair comparison would seem to require using objects belonging to R's reference class (type '?setRefClass' at the R prompt for documentation on this class); reference class objects more closely resemble those we see in Python. In what follows I take this path, but only after first seeing how R's generic list data structure performs relative to Python's list data structure.

The runtime difference between copying, or passing by value, and passing by reference is made concrete when we compare the output from the R function in Figure 1 with the output from the Python function in Figure 2. Both code fragments compute the powerset of a set of elements, a task for which a recursive approach is well-suited. Both code fragments have the same basic structure. As far as I know, that structure does not favor one language over the other. If this is true, then the difference in runtimes that we see can be attributed almost entirely to the difference in the data structures involved and whether the corresponding object is passed by value or passed by reference.

When there are 19 elements in the argument list, R needs 6.42 seconds to complete the task whereas Python needs less than 0.62 seconds, or around one-tenth of the time. The powerset of a 19-element list contains more than 524K members.

When there are 26 elements in the argument list, Python completes the task in just under 1 minute and 25 seconds. By contrast, R needs almost 17 minutes on my computer---nearly a 12-fold increase in runtime. The powerset for a 26-element list contains over 67 million members.

Clearly, Python's list data structure is far better suited to this task than R's list data structure. We will see in a moment that the additional time needed by R cannot be attributed solely to the fact that R is copying the list argument each time get_subsets is called.

Is there a minor adjustment that might improve the runtime of the R code? The list-append used in Figure 1 has its cost, but it is far better than the approach used in Figure 3. The runtime of the code in Figure 3 is much worse due to the extra copying of *pwrset* that occurs inside the for-loop. Without resorting to more advanced techniques or abandoning the code structure used in both Figures 1 and 2 (a code structure I want to keep the same across the two languages in order to satisfy the "all else being equal" constraint), I cannot readily think of a way in R to write function *get_subsets* that improves upon the code in Figure 1.

Figure 1: R code for generating the powerset of a list of elements

```
# R code for generating the powerset of a vector of ints or strings:

get_subsets <- function(pwrset, lst) {
  # When first called, 'pwrset' is an empty list.  'lst' is a vector of
  # elements (can be ints or strings) for which we need a powerset.
  # get_subsets returns a list which is the powerset of lst.
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets(list(), lst[-1])
    # the idea in what follows is to insert 'elem' into every set
    # in remainder, append these new sets to pwrset, and then
    # append the remaining sets in remainder to pwrset.

    # the minimum length for remainder will be 2
    for(i in 1:length(remainder)){
      val <- unlist(remainder[[i]])
      pwrset[[length(pwrset) + 1]] <- c(elem, val)
      ifelse(is.null(val), pwrset[[length(pwrset) + 1]] <- list(),
             pwrset[[length(pwrset) + 1]] <- val)
    } # end of for-loop, index i
  } # end of if stmt

  if(length(lst)== 1){
    pwrset[[length(pwrset) + 1]] <- lst[1]
    pwrset[[length(pwrset) + 1]] <- list()
  }

  if(length(lst)== 0){
    pwrset[[length(pwrset) + 1]] <- list()
  }
  return(pwrset)
}

gen_powerset <- function(lst){
  return(get_subsets(list(), lst))
}

start = Sys.time()
nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s', ","))
length(nlst)
# 19
ans <- gen_powerset(nlst)
length(ans)
#  524288
stop = Sys.time()
round(stop - start, 2)
# Time difference of 6.42 secs
```

Figure 2: Python code for generating the powerset of a list of elements

```
# Python code for generating the powerset of a list of elements:

def getSubsets(pwrset, lst):
    """ When first called, pwrset will be an empty list.  With each recursive
    call, pwrset accumulates the sets we need.  'lst' is the list that remains
    for which we need to get a powerset.
    Return value is the powerset of lst. """
    if len(lst) > 1:
        elem = lst[0]
        remainder = getSubsets([], lst[1:])

        index = 0
        for nextlist in remainder:
            lstcopy = list(nextlist[:])
            lstcopy.append(elem)
            pwrset.append(lstcopy)
            pwrset.append(remainder[index])
            index += 1
        return pwrset
    elif len(lst) == 1:
        if type(lst[0]== int):
            pwrset.append(lst)
        else:
            pwrset.append(list(lst[0]))
        pwrset.append([])
        return pwrset
    else:
        pwrset.append([])
        return pwrset

def genPowerset(lst):
    """ lst is a list of elements (can be ints, strings, or tuples)
    genPowerset returns the powerset of lst."""
    return getSubsets([], lst)


from datetime import datetime
start = datetime.now()
nlst = str.split('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s', ",")
print(len(nlst))
# 19
ans = genPowerset(nlst)
print(len(ans))
#  524288
stop = datetime.now()
print("Time difference of", stop - start)
# Time difference of 0:00:00.615824
```

Figure 3: Expensive R code for generating the powerset of a list of elements. Additional copying happens in the for-loop, copying not found in the code of Figure 1.

```
get_subsets02 <- function(pwrset, lst) {
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets02(list(c()), lst[-1])

    for(i in 1:length(remainder)){
      val <- remainder[[i]]
      pwrset <- c(pwrset, list(c(elem, val)))
      if(!is.null(val)) { pwrset <- c(pwrset, list(val)) }
    }
  } # end of if stmt

  if(length(lst)== 1){
    pwrset <- c(pwrset, list(lst[1]))
  }
  return(pwrset)
}

gen_powerset02 <- function(lst){
  return(get_subsets02(list(c()), lst))
}


start = Sys.time()
nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p', ","))
length(nlst)
# 16
ans <- gen_powerset02(nlst)
length(ans)
# [1] 65536
stop = Sys.time()
round(stop - start, 2)
# Time difference of 24.78 secs
```

Might we see meaningful gains in the runtime for the R code if we create a reference class object to pass to get_subsets? An initial attempt at this is shown in Figure 4. The mList object in Figure 4 is passed by reference.

Comparing the runtime for the code in Figure 1 with that of the code in Figure 4, we see that the *mList* reference class object does *not* improve the runtime. The performance is instead much worse. The deep assignment operator, "<<-", in the method, *append*, apparently has to do a lot of work searching back through the list of parent environments created with each function call to make the assignment to *pwrset*. Since reference class objects are unfamiliar territory for me, I am not going to try to improve upon the code in Figure 4 but will instead look at another method that Hadley Wickham suggests (Chapter 8 of *Advanced R*).

Figure 5 shows the code for this other method. Creating a list in its own environment is a vast improvement over *mList* in Figure 4. However, the code in Figure 5 does not yield any significant improvement over the *original* approach taken in Figure 1.

Figure 4: Using a reference class object in R. The object does not get copied when passed to the function that generates the powerset of a list of elements

```
mList <- setRefClass("mList",
                     fields = list(members = "list"))
mList$methods(
  append = function(val){
    members <<- c(members, list(val))
  }
)

get_subsets <- function(pwrset, lst) {
  # get_subsets returns a list which is the powerset of lst.
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets(mList$new(), lst[-1])

    for(i in 1:length(remainder$members)){
      val <- unlist(remainder$members[[i]])
      pwrset$append(c(elem, val))
      ifelse(is.null(val), pwrset$append(c()),
             pwrset$append(val))
    } # end of for-loop, index i
  } # end of if stmt

  if(length(lst)== 1){
    pwrset$append(lst[1])
    pwrset$append(c())
  }
  if(length(lst)== 0){
    pwrset$append(c())
  }
  return(pwrset)
}

gen_powerset <- function(lst){
  return(get_subsets(mList$new(), lst))
}


start = Sys.time()
nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o', ","))
length(nlst)
# 15
ans <- gen_powerset(nlst)
length(ans$members)
#  [1] 32768
stop = Sys.time()
round(stop - start, 2)
# Time difference of 9.41 secs
```

Figure 5: Using an environment list in R for improved performance. (Function *gen_subsets* not shown.)

```
lstenv <- new.env(parent = emptyenv())
lstenv$empty.list <- list()


gen_powerset <- function(lst){
  return(get_subsets(lstenv$empty.list, lst))
}

start = Sys.time()
nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s', ","))
length(nlst)
# 19
ans <- gen_powerset(nlst)
length(ans)
# [1] 524288
stop = Sys.time()
round(stop - start, 2)
# Time difference of 6.05 secs (vs. 6.42 seconds in Figure 1)



start = Sys.time()
nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,
                        m,n,o,p,q,r,s,t,u,v,w,x,y,z', ","))
length(nlst)
# 26
ans <- gen_powerset(nlst)
length(ans)
# [1] 67108864
stop = Sys.time()
round(stop - start, 2)
# Time difference of 16.35 mins (whereas Python needs only
# about 1.42 minutes to complete this task)
```

While there are surely other approaches we can explore in R, it is likely we will need to do something relatively sophisticated to get R to compute the powerset of a set in a runtime that is comparable to Python's. In other words, it does look like we will have to come up with a solution in R that is conceptually complex relative to the solution shown in Figure 1 if we are to see a gain in computational efficiency that puts R on a par with Python for this kind of recursive task.

Why does Python do such a better job with this recursive task when the code in Figure 2 is no more complex syntactically than the code in Figure 1? The greater complexity lies below the surface with Python's list data structure. There is an additional layer of abstraction in the 'pwrset.append()' statement that is not found in the 'pwrset[[*index*]] <- val' statement. We see the assignment operator and the indexing in the latter, but not in the former. When pwrset.append(val) is executed, the assignment and indexing of the pwrset list object is happening in the list object's append method found in its class definition. It is this added layer of abstraction that provides us with a way to pass the list by reference rather than "by value". Of course, a lot more is going on below the

surface than just this; otherwise we would see the R code in Figures 4 and 5 have runtimes closer what we see for the Python code. What is under the hood in Python is designed to work with the list object we use on the surface. I am not so sure that the same can be said for the R data structures I employ in Figures 4 and 5. Perhaps some of Python's speed advantage is also due to better memory management techniques.

Iterative Runtime Comparison

In R I have learned to replace for-loops with hash tables (Python's equivalent of a dictionary) whenever possible. The decrease in runtimes can be several orders of magnitude. But Figures 6 and 7 suggest that R handles for-loops much more efficiently than Python does.

Figure 6: R code for runtime comparison of for-loop code.

```
start <- Sys.time()
n <- 10000
p <- 10000
count <- 0

for(ii in 1:10){
  for(i in 1:n){
    for(j in 1:p) count <- count + 1
  }
}
stop <- Sys.time()
ans <- round(as.double(difftime(stop, start, units= 'secs'))/10, 3)
print(paste("Average time difference of", ans, "seconds."))
# [1] "Average time difference of 4.487 seconds."
```

Figure 7: Python code for runtime comparison of for-loop code.

```
from datetime import datetime
start = datetime.now()
n = 10000
p = 10000
count = 0
for index in range(10):
    for i in range(n):
        for j in range(p):
            count += 1
stop = datetime.now()
delta = stop - start
delta_secs = delta.seconds
delta_micro = delta.microseconds/1000000

time_diff = round((delta_secs + delta_micro)/10, 3)
print("Average time difference of " + str(time_diff) + " seconds.")
# Average time difference of 11.868 seconds.
```

The advantage that R has with for-loops, however, is completely erased in the next test. Compare the runtime of the code in Figure 8 with the runtime of the code in Figure 9. Strings in R are character vectors, but ones in which the individual characters of the string cannot be referenced, or accessed, using an index. For example, if mystr = 'abcd', mystr[3] will return NA rather than 'c'. By contrast, strings in Python are lists. In Python, mystr[0] returns 'a'. Once again, the difference in data structures makes all the difference in runtimes. Python's superior speed ($1/50^{th}$ that of R's) is extraordinary given R's apparent advantage walking through for-loops.

Figure 8: R code---string concatenation inside a for-loop.

```
start <- Sys.time()
n <- 100000
mystring <- ''

for(i in 1:n){
  ifelse(i != 1, mystring <- paste(mystring, i, sep=","), mystring <- "1")
}
substring(mystring, 1, 30)
# [1] "1,2,3,4,5,6,7,8,9,10,11,12,13,"
substring(mystring, nchar(mystring) - 50)
# [1] "92,99993,99994,99995,99996,99997,99998,99999,100000"
print(paste("The length of the string is: ", nchar(mystring)))
# [1] "The length of the string is:  588894"
stop <- Sys.time()
round(stop - start, 2)
# Time difference of 1.67 mins
```

Figure 9: Python code--- string concatenation inside a for-loop.

```
start = datetime.now()
n = 100000
mystring = ""
for i in range(n):
    if (i + 1) != n:
        mystring = mystring + str(i + 1) + ","
    else:
        mystring = mystring + str(i + 1)

print(mystring[:30])
print(mystring[(len(mystring) - 51):])
print("The length of the string is: " + str(len(mystring)))
stop = datetime.now()
print("Time difference of", stop - start)
# 1,2,3,4,5,6,7,8,9,10,11,12,13,
# 92,99993,99994,99995,99996,99997,99998,99999,100000
# The length of the string is: 588894
# Time difference of 0:00:01.983526
```

Takeaway from the comparison

Although the sample size is quite small, the above comparisons suggest that Python is computationally more efficient than R. So if a lot of programming is required to prepare the dataset one needs for analysis, perhaps because millions of records need to be processed to bring this dataset into being and/or because data has to be pulled in from multiple sources, Python is likely to be the better choice. The reasons for choosing Python are even stronger if the processing to produce such a dataset has to be done on a regular basis---as in the case of, say, monthly reporting. For me this is noteworthy because I am an R enthusiast and find R to be an excellent tool for data analysis. In fact, in many instances it is arguably the better tool when quickness and ease are important (e.g., dplyr and pipes). Which language is the better one to use, however, depends on the task at hand and where the priority lies---whether with speed, the simplicity/ease of producing the code to execute the task, or some other desideratum.