G. Schaefer

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 (copy-on-modify, also sometimes referred to as “passed by value”), 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, one would expect Python to outperform R for many recursive tasks. Figures 1 and 2 below provide one example of just how much this difference can affect performance. We might try to level the playing field, then, by using R reference class objects (type ‘?setRefClass’ at the R prompt for documentation on this class) or R6 objects (r6.r-lib.org) since both of these object-oriented approaches more closely resemble the objects we see in Python. To my surprise, however, and as we see in what follows, these other approaches do not improve upon R’s performance for the recursive task at hand.

* * * * *

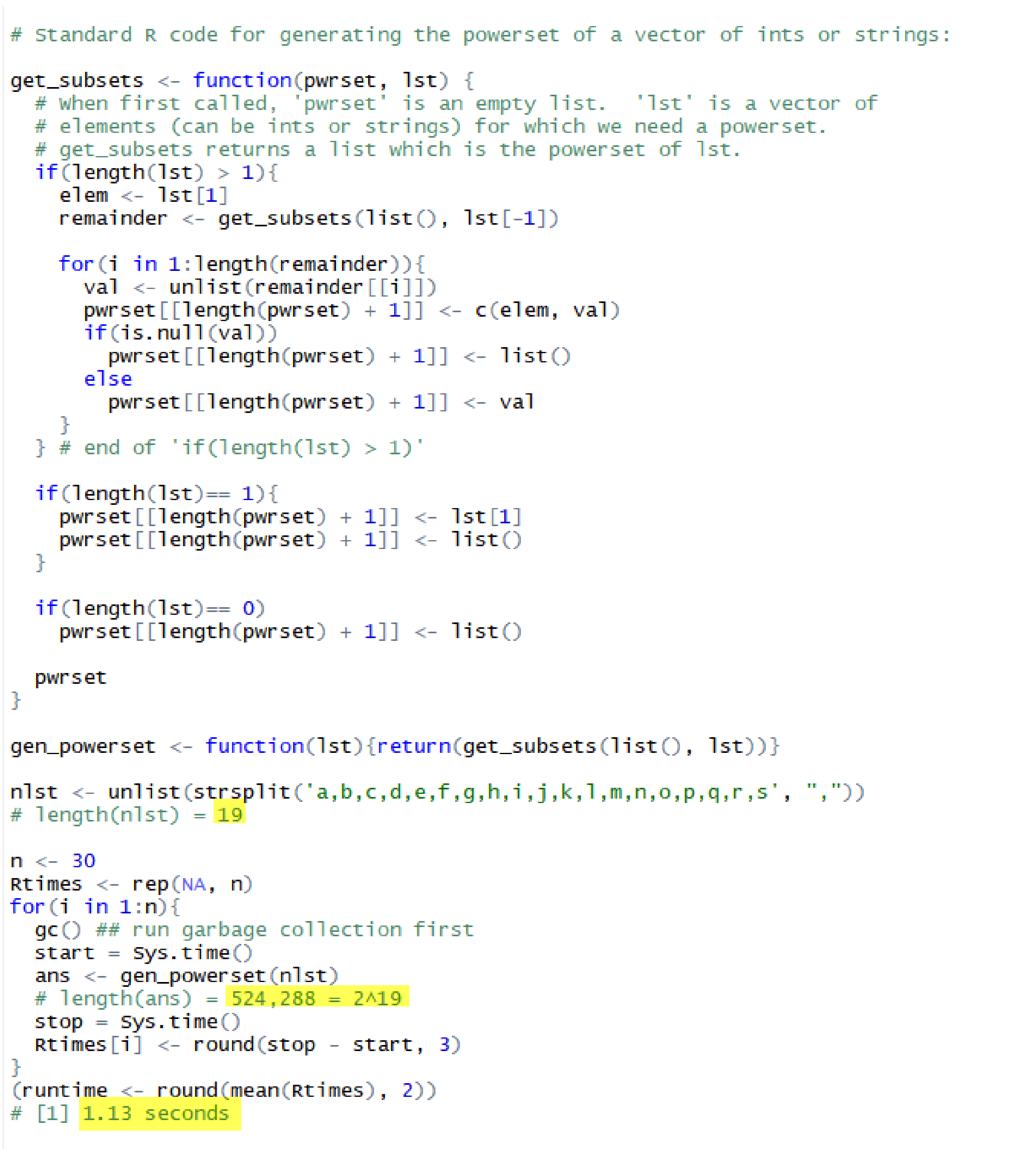
The runtime difference between copy-on-modify semantics and passing by reference is made concrete when we compare the performance of the R code in Figure 1, which uses a generic list data structure, with the performance of the Python code in Figure 2, which also employs a list data structure. Both code 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 logical structure. Thus, the difference in runtimes that we see should be due more to differences in the data structures being used (e.g., how they exist in memory) and whether objects are “passed by value” or passed by reference on function calls.

When there are 19 elements in the argument list, R needs 1.13 seconds to complete the task whereas Python needs less than 0.63 seconds--- a 44% drop in runtime. The powerset of a 19-element list contains more than 524K members (precisely 2^{19}). In the R code, if we replace the if . . . else construct in the for-loop with an ifelse statement, the runtime increases to 5.71 seconds, showing just how expensive R’s ifelse function can be.

When there are 26 elements in the argument list, Python completes the task in just under 1 minute and 25 seconds. The powerset for a 26-element list contains over 67 million members. The final object size in Python is around 604MB. By contrast, R needs 4.34 minutes on my computer---more than a 3-fold increase. And the object size is around 12.3GB (using pryr::object_size(*object*); RStudio says the object is around 57GB, which cannot be right since I have only 32GB of RAM).

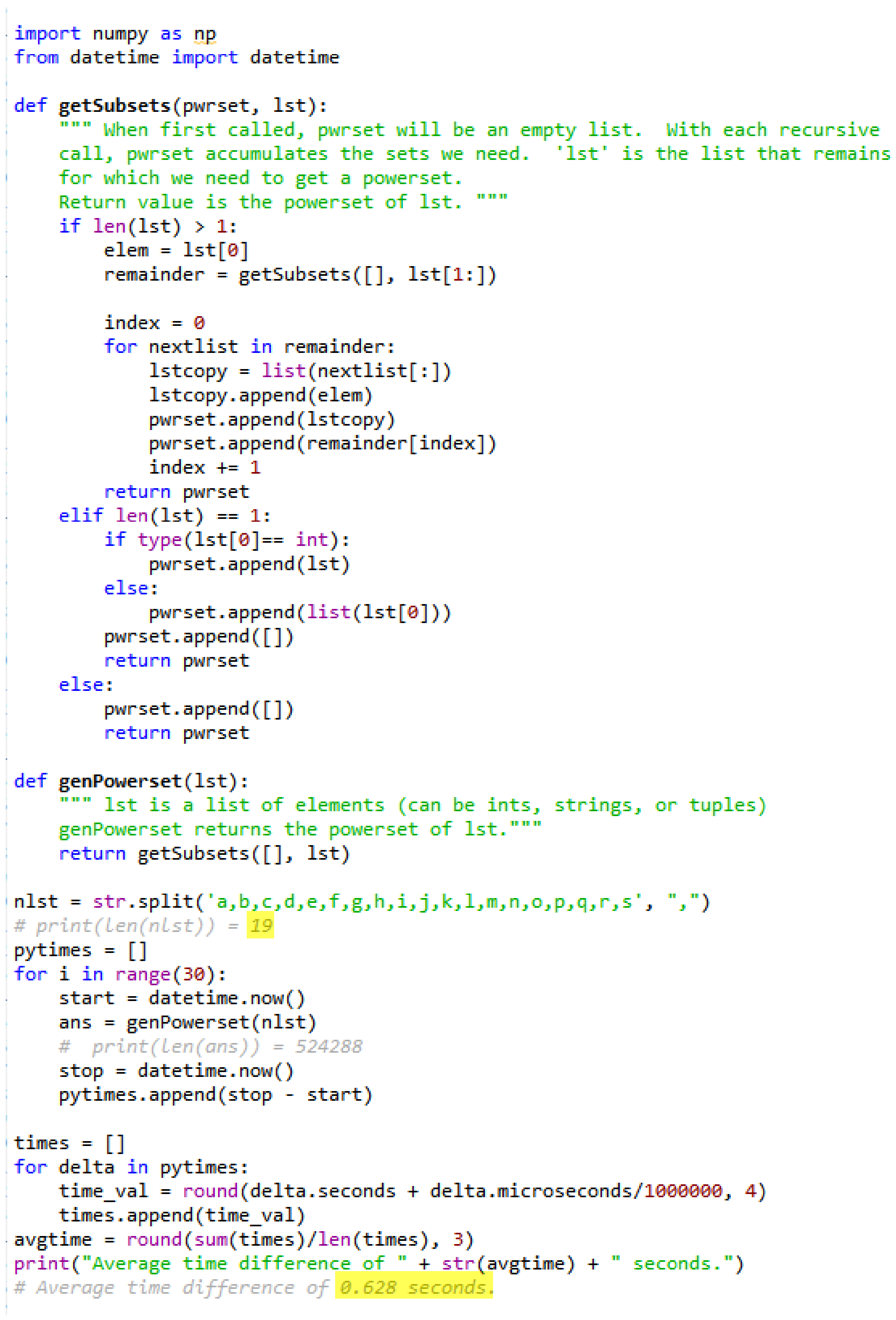
Clearly, Python's list data structure is far better suited to this task than R's list data structure. We will see in a moment, with the examples using reference classes and R6 classes, that the additional time needed by R cannot be attributed solely to the fact that R is copying the list argument each time get_subsets is called.

Figure 1: Typical approach in R for generating the powerset of a list of elements



```r
# Standard 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])

    for(i in 1:length(remainder)){
      val <- unlist(remainder[[i]])
      pwrset[[length(pwrset) + 1]] <- c(elem, val)
      if(is.null(val))
        pwrset[[length(pwrset) + 1]] <- list()
      else
        pwrset[[length(pwrset) + 1]] <- val
    }
  } # end of 'if(length(lst) > 1)'

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

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

  pwrset
}

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

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

n <- 30
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc() ## run garbage collection first
  start = Sys.time()
  ans <- gen_powerset(nlst)
  # length(ans) = 524,288 = 2^19
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1] 1.13 seconds
```

<u>Figure 2</u>: Python code for generating the powerset of a list of elements.



```python
import numpy as np
from datetime import datetime

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)

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
pytimes = []
for i in range(30):
    start = datetime.now()
    ans = genPowerset(nlst)
    #  print(len(ans)) = 524288
    stop = datetime.now()
    pytimes.append(stop - start)

times = []
for delta in pytimes:
    time_val = round(delta.seconds + delta.microseconds/1000000, 4)
    times.append(time_val)
avgtime = round(sum(times)/len(times), 3)
print("Average time difference of " + str(avgtime) + " seconds.")
# Average time difference of 0.628 seconds.
```

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 below in Figure 3. The runtime of the Figure 3 code is much worse due to the extra copying of *pwrset* that occurs inside the for-loop.

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.

```r
# R code for very inefficient powerset function:

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]))

  pwrset
}

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


nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p', ","))
length(nlst)
# 16

n <- 10
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc() ## run garbage collection first
  start = Sys.time()
  ans <- gen_powerset02(nlst)
  # length(ans) = 65,536
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1] 24.3 seconds
```

The remaining figures, or panels, of this section show some less conventional approaches in R to finding the powerset of a set. None of these other approaches improve upon the code in Figure 1.

Figure 4 shows one way to use a reference class for the powerset recursive task; the *mList* object is passed by reference.

Comparing the performance of the Figure 1 code and the Figure 3 code with that of the Figure 4 code, we see that the *mList* reference class object significantly *degrades* performance. The deep assignment operator, "<<-", in the method, *append*, apparently is working hard searching back through the list of parent environments created with each function call in order to make the assignment to *pwrset*.

Figure 4: Using a reference class object in R. The object does not get copied when passed to get_subsets.

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

get_subsets <- function(pwrset, 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$append(c(elem, val))
      if(is.null(val))
        pwrset$append(c())
      else
        pwrset$append(val)
    }
  }
  if(length(lst)== 1){
    pwrset$append(lst[1])
    pwrset$append(c())
  }
  if(length(lst)== 0)
    pwrset$append(c())

  pwrset
}

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

nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p', ","))
# length(nlst) = 16

n <- 10
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc() ## run garbage collection first
  start = Sys.time()
  ans <- gen_powerset(nlst)
  # length(ans$members) = 65,536 = 2^16
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1] 33.56 seconds
```

If I tweak the code in Figure 4 by replacing the for-loop in get_subsets with *lapply*, the result is a significant further decline in performance. This is shown in Figure 5. If I take push_element out of get_subsets and use *mapply* instead of *lapply*, performance remains the same as in Figure 5.

Figure 5: Using an R reference class object in combination with *lapply*.

```r
get_subsets <- function(pwrset, lst){
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets(mList$new(), lst[-1])

    push_element <- function(remainder_member){
      val <- unlist(remainder_member)
      pwrset$append(c(elem, val))
      if(is.null(val))
        pwrset$append(c())
      else
        pwrset$append(val)
    }

    lapply(remainder$members, push_element)
  }
  if(length(lst)== 1){
    pwrset$append(lst[1])
    pwrset$append(c())
  }
  if(length(lst)== 0)
    pwrset$append(c())

  pwrset
}

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

nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p', ","))
# length(nlst) = 16

n <- 10
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc() ## run garbage collection first
  start = Sys.time()
  ans <- gen_powerset(nlst)
  # length(ans$members) = 65,536 = 2^16
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1]  40.96 seconds
```

Figure 6 shows code for another method that Hadley Wickham suggests (Chapter 8 of *Advanced R*)---using an environment as our data structure. While this is a vast improvement over the reference class approach, the code in Figure 6 does not improve upon the performance of the original, “standard”, approach in Figure 1.

Figure 6: Using an environment list in R.

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

get_subsets <- function(pwrset, lst){
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets(lstenv$empty.list, lst[-1])
    for(i in 1:length(remainder)){
      val <- unlist(remainder[[i]])
      pwrset[[length(pwrset) + 1]] <- c(elem, val)
      if(is.null(val))
        pwrset[[length(pwrset) + 1]] <- list()
      else
        pwrset[[length(pwrset) + 1]] <- val
    }
  }
  if(length(lst)== 1){
    pwrset[[length(pwrset) + 1]] <- lst[1]
    pwrset[[length(pwrset) + 1]] <- list()
  }
  if(length(lst)== 0)
    pwrset[[length(pwrset) + 1]] <- list()

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

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
n <- 30
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc(); start = Sys.time()
  ans <- gen_powerset(nlst)
  # length(ans) = 524,288 = 2^19
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1] 1.27 seconds


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
start = Sys.time()
ans <- gen_powerset(nlst)
# length(ans) = 67108864
stop = Sys.time()
round(stop - start, 2)
# Time difference of 4.42 mins (whereas Python needs only
# about 1.42 minutes to complete this task)
```

Figure 7 shows performance of R when using an R6 class. R6 classes are faster than R's reference classes but, for this recursive task, still significantly slower than the code of Figures 1 and 6.

Figure 7: When using an R6 class, we see about a 20% performance gain over the R reference class. Even so, the performance is dismal compared to the R code in Figure 1.

```
require(R6)
tmpList <- R6Class("tmpList", list(
  members = list(),
  append = function(val){
    self$members <- c(self$members, list(val))
    invisible(self)
  }
))
get_subsets <- function(pwrset, lst){
  if(length(lst) > 1){
    elem <- lst[1]
    remainder <- get_subsets(tmpList$new(), lst[-1])
    for(i in 1:length(remainder)){
      val <- unlist(remainder[[i]])
      pwrset$append(c(elem, val))
      if(is.null(val))
        pwrset$append(c())
      else
        pwrset$append(val)
    }
  }
  if(length(lst)== 1){
    pwrset$append(lst[1])
    pwrset$append(c())
  }
  if(length(lst)== 0)
    pwrset$append(c())

  pwrset$members
}

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

nlst <- unlist(strsplit('a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p', ","))
# length(nlst) = 16

n <- 10
Rtimes <- rep(NA, n)
for(i in 1:n){
  gc(); start = Sys.time()
  ans <- gen_powerset(nlst)
  # length(ans) = 65,536 = 2^16
  stop = Sys.time()
  Rtimes[i] <- round(stop - start, 3)
}
(runtime <- round(mean(Rtimes), 2))
# [1] 26.82 seconds; almost 20% faster than reference classes, but
# much slower than using an environment list data structure
```

While there are surely other approaches we can explore in R, it is likely we will need to employ some advanced techniques to get R to compute the powerset of a set in a runtime that is comparable to Python's.

Why does Python do such a better job on this task when the code in Figure 2 is no more complex syntactically than the code in Figure 1? One reason for Python's superior performance likely has to do with the fact that its list data structure is part of the original design, so-to-speak. By contrast, R's reference classes are an extension of the S4 classes, and the R6 classes are, loosely-speaking, an extension of S3; in both cases we have something constructed from, or sitting on top of, something else rather than being the direct object, or aim, of the initial design of the language. Since R was not designed for speed at its inception, getting R to run faster would seem to require that certain parts of it be re-designed from the ground up.

Finally, it may also be that some of Python's speed advantage is due to better memory management techniques. As noted above, in Python the powerset of a 26-element list required 604MB of memory, whereas in R the same object required 20 times as much memory.

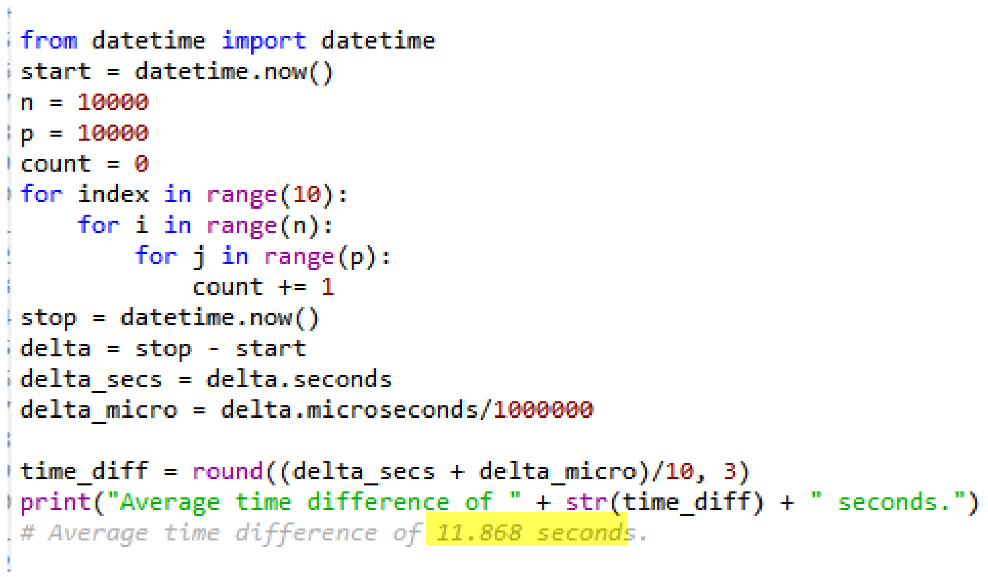
Iterative Runtime Comparison

Figures 8 and 9 suggest that R handles for-loops much more efficiently than Python does.

Figure 8: 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."
```

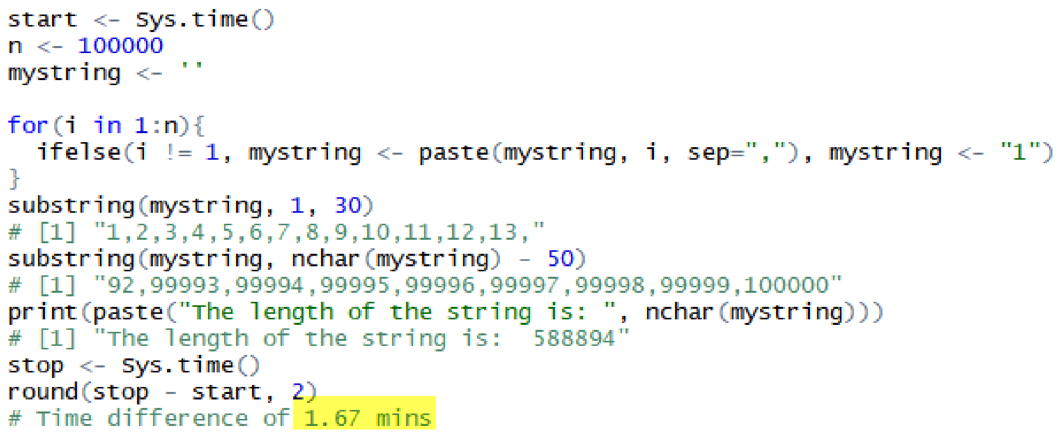
Figure 9: Python code for runtime comparison of for-loop code.



```
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.
```

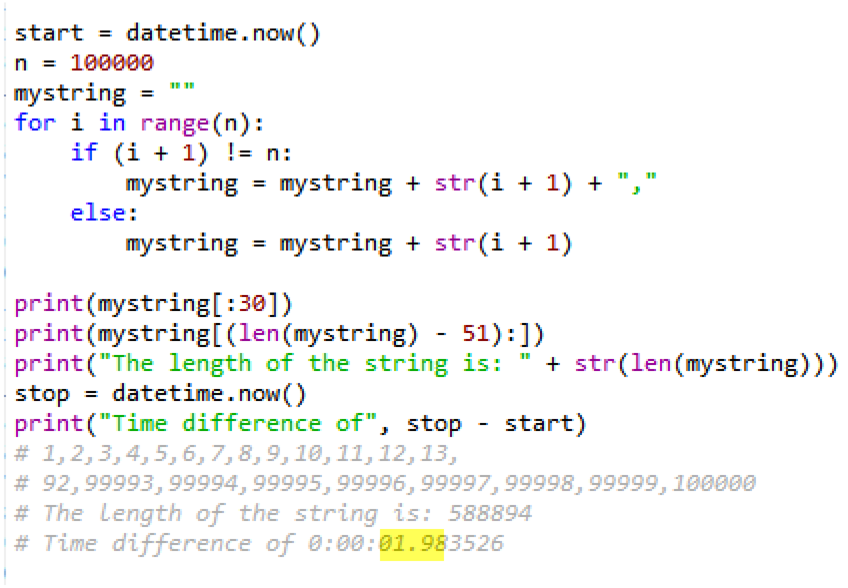
The advantage that R has with for-loops, however, is completely erased in the next test. Compare the runtime of the code in Figure 10 with the runtime of the code in Figure 11. Strings in R are character vectors, but ones in which the individual characters of the string cannot be referenced, or accessed, 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 (1/50th that of R's) is extraordinary given R's apparent advantage walking through for-loops.

Figure 10: 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 11: 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 noticeably faster 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 or weekly reporting. With our powerset task it is worth reiterating that the version of Python I have installed needed 1.42 minutes to compute the powerset of a 26-element set, whereas my version of R ran three times longer, clocking in at 4.34 minutes.

Still, it won't always be the case that Python is the better choice. R is an excellent tool for data analysis. In many instances it is likely to be the better tool when quickness and ease are important. Thus, which language is the better one to use 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, displaying data, or some other desideratum.