R or Python?

For the statistician and/or data scientist equally fluent in both R and Python, it is worth knowing when one language is truly better than the other for a given task, or set of tasks, and why this is so. For tasks other than data analysis and data wrangling, the language to use---if the choice is only between these two---is almost certainly Python.

That is because R is designed primarily for data analysis and working with data, whereas Python is a much more general-purpose programming language. Insofar as Python is well-suited for a wider range of programming tasks, it is the more powerful language of the two. But this versatility, this lack of specialization, comes with an additional cost: to truly leverage the power of Python for data analysis, one has to feel comfortable working with more types of objects (e.g., not just with functions but with classes). This can mean that manipulating data in Python is sometimes not quite as straightforward, efficient, or intuitive as in R. (One way to test this admittedly evaluative statement would be by comparing the number of statistics departments primarily using Python in their courses versus those primarily using R. The preferred language will almost certainly be R, and not just be for historical reasons, i.e., due to inertia. Within data science departments, on the other hand, where there is a much greater focus on computer algorithms and machine learning with big data, and thus an overlap with computer science departments, there is a better chance that the preferred language will be Python.)

While both R and Python are object-oriented languages, the types of objects they rely on differ greatly. Python programmers frequently rely on class objects for purposes of encapsulation, so much so that the data analyst using Python will want to know how these objects work, how to define them, how to call their methods, and so forth. Analogous objects in R are the reference and R6 classes. But these objects are extensions to the language, not intrinsic to it. Most data analysts using R are likely to never have a need for them. (Those working in bioinformatics seem to be the primary set of users for reference classes---see bioconductor.org. Since the newer R6 objects are more efficient, one should avoid reference classes if at all possible. A performance comparison between the two kinds of objects is found below.)

The object in R that has clear priority is the function. For data analytic work, nearly everything one needs to do can be done in R by writing functions. Compared to Python, then, R is more of a functional programming language.

The differences between the two languages are such that there will be certain data analysis projects for which, arguably, one or the other of the languages will be the better choice. If a project involves a great deal of code that is going into production and the code has to run on “big data”, Python is likely to be the better choice since it is faster than R for certain basic tasks. We can get a sense of just how much faster by looking at two examples, one recursive, the other purely iterative. While our projects may rarely rely on recursion in so direct a manner as in the example that follows, this recursive task tests the efficiency (or inefficiency) of passing an argument to a function. As for the iterative task, for-loops and strings see widespread use.

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 that retrieves from, or inserts 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.

Neither R nor Python are designed to efficiently handle recursion. But R is especially disadvantaged in this respect since functions calls in R handle arguments on a copy-on-modify basis: if the argument to a function is modified within the function, it is first copied. In Python, by contrast, arguments are passed by reference, meaning that if an argument object is altered in the function called, the change made to the object persists when we return to the parent environment. Because copying is computationally expensive, one would expect Python to easily outperform R for recursive tasks, especially as the objects passed grow larger in size. Figures 1 and 2 below show just how great the difference can be.

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 class objects allow arguments to be passed by reference. To my surprise, however, and as we see in what follows, these other approaches result in significantly worse performance than what the code in Figure 1 yields.

* * * * *

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 in certain contexts.

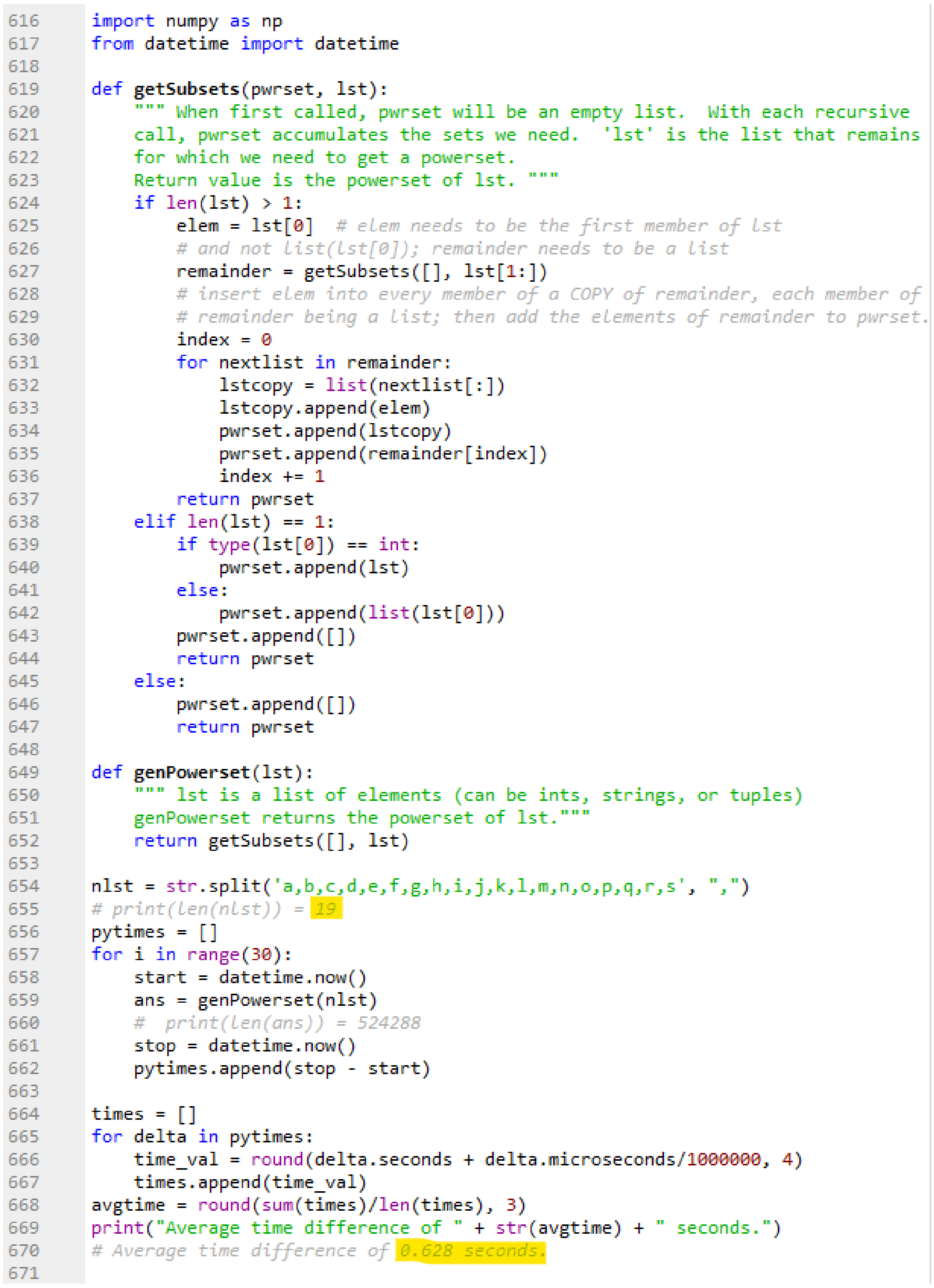
When there are 26 elements in the argument list, Python completes the task in just under 1 minute and 25 seconds. The powerset of a 26-element list contains over 67 million members. The final object size in Python is around 604MB. By contrast, R needs 4.34 minutes on my computer---more than a 3-fold increase. And the object size is around 12.3GB (using pryr::object_size(*object*); RStudio says the object is around 57GB; if this is correct, then nearly all of both physical and virtual memory on my machine are needed to handle this object).

Such a great difference in object size suggests that Python's list data structure is far better suited to this task than R's list data structure. The code in Figures 4, 5, and 7 shows 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. Thus, it must be the case that how these list data structures are implemented in memory also clearly matters.

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
```

Figure 2: 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]  # elem needs to be the first member of lst
        # and not list(lst[0]); remainder needs to be a list
        remainder = getSubsets([], lst[1:])
        # insert elem into every member of a COPY of remainder, each member of
        # remainder being a list; then add the elements of remainder to pwrset.
        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 rebuilding the list as 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 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 Figure 4, 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*. If we try to avoid the concatenation on the right-hand side of mList's append method by adding an index to the members list (in other words, if we try to mimic the assignment code in Figure 1), the performance of the code degrades even further.

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
```

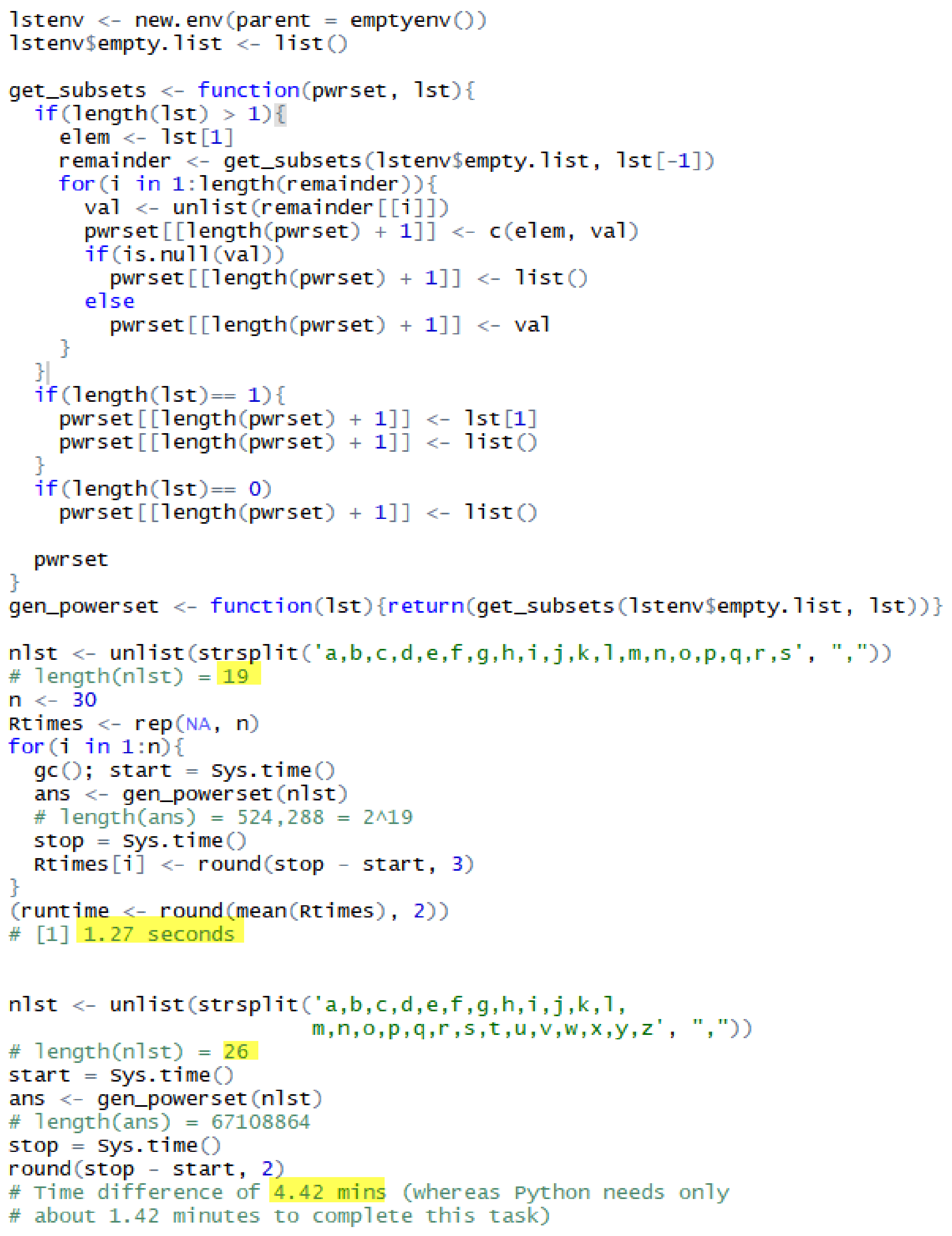
Tweaking the code in Figure 4 by replacing the for-loop in get_subsets with *lapply* also results in a significant further decline in performance. This is shown in Figure 5. If I take push_element out of get_subsets and use *mapply* instead of *lapply*, performance remains the same as in Figure 5.

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

```
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.



```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 R's performance 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.

```r
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 appears one will need to do something pretty special, programming-wise, to get R to compute the powerset of a set in a runtime that is comparable to Python's. It won't do, for the purpose of comparing the performance of R with that of Python, to write the recursive function in C++ and then make use of the Rcpp package in R, since we would no longer be comparing Python with R but rather Python with C++ code that has an R interface. (In my experience Rcpp is seldom a solution to performance issues because of the difficulty of identifying chunks of code doing a single, specific task that can be completely isolated from the rest of one's code. One cannot move in and out of R while the C++ function is doing its work. The section of code that we identify also has to be such that it is creating a bottleneck in the performance of our R code. Since R is already very efficient with many of the tasks the data analyst so often relies upon, finding places in one's code where Rcpp will boost performance is not especially easy.)

Python has a remarkable tool of its own for increasing the performance of the code in Figure 2 above. It is more versatile than Rcpp and far easier to apply. This tool is the Cython compiler, which can convert Python code into C code. If we *cythonize* the 2 functions in Figure 2 (something that requires no re-write of the code whatsoever), we will see a performance improvement of over 30% on the 26-member set: after cythonizing, the time it takes to construct the more than 67 million element powerset is reduced from 77.1 seconds to 53.2 seconds. Python also has other tools to improve the performance of recursive tasks. If the recursion relies on results already computed (as a recursive program for computing the Fibonacci sequence would), we can vastly---and easily--- improve performance via memoization using the functools.lru_cache decorator. (See Luciano Ramalho's discussion of this tool in his excellent book, *Fluent Python*. On p. 170 of his book he also refers us to Alexey Kachayev's *fn.py* package: "It includes a `@recur.tco` decorator that implements tail-call optimization for unlimited recursion in Python, among other functions, data structures, and recipes". I have not dug into the *fn.py* package yet and so am unable to say anything about how often we might be able to make use of it.)

Why does Python do such a better job building the powerset of a set? The answer is not due solely to the difference in how arguments are passed in function calls. If that were true, R's performance would be more on a level with Python's when we make use of an R6 class. Copy-on-modify versus by-reference is one important difference between the two languages. But another, clearly, is that of data structure design---the overhead of the data structure, i.e., how that data structure exists in memory. While Python's list data structure is part of the original design of the language, 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, reference class and R6 class, we have an object constructed from, or sitting on top of, another type of object or set of objects rather than being central to the original design of the language. As the results in Figure 7 show, the R6 class used there is no match for Python's list data structure: the former needs nearly 27 seconds to accomplish what the latter can do in less than half a second.

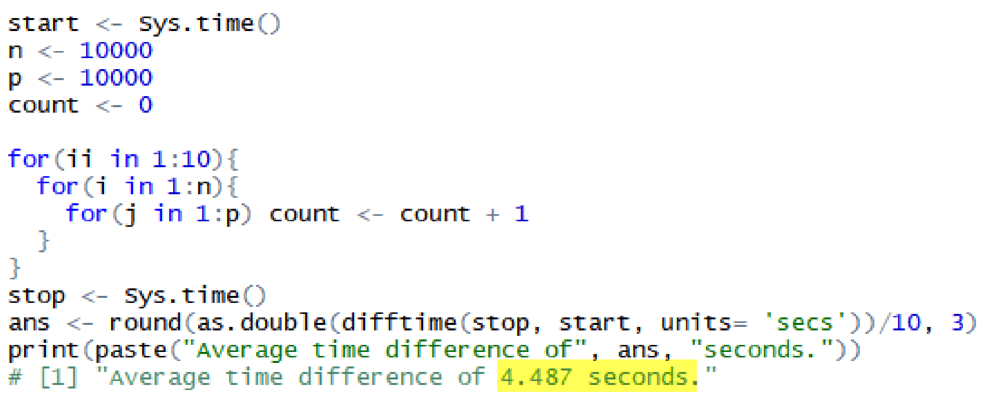
Interestingly, the performance results of the code in Figures 4-7 lead one to have a far greater appreciation for what standard R code (that used in Figure 1) can do. One should not have to resort to extraordinary methods to do simple tasks. R shines in this respect. Recursive tasks, however, are not something that R is designed for. So if a recursive routine lies within the context of a larger R program, and can be completely isolated from it, we would want to translate it into C++ code and then call the C++ code from within the surrounding R code using the Rcpp functions. But while some R performance bottlenecks can be addressed in this way, others we may have to live with. R's fantastic user-friendliness sometimes comes at a cost.

Finally, some of Python's speed advantage is also likely 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. I would be surprised if this difference in object size were due entirely to a difference in the overhead R needs to manipulate its list data structure.

Iterative Runtime Comparison

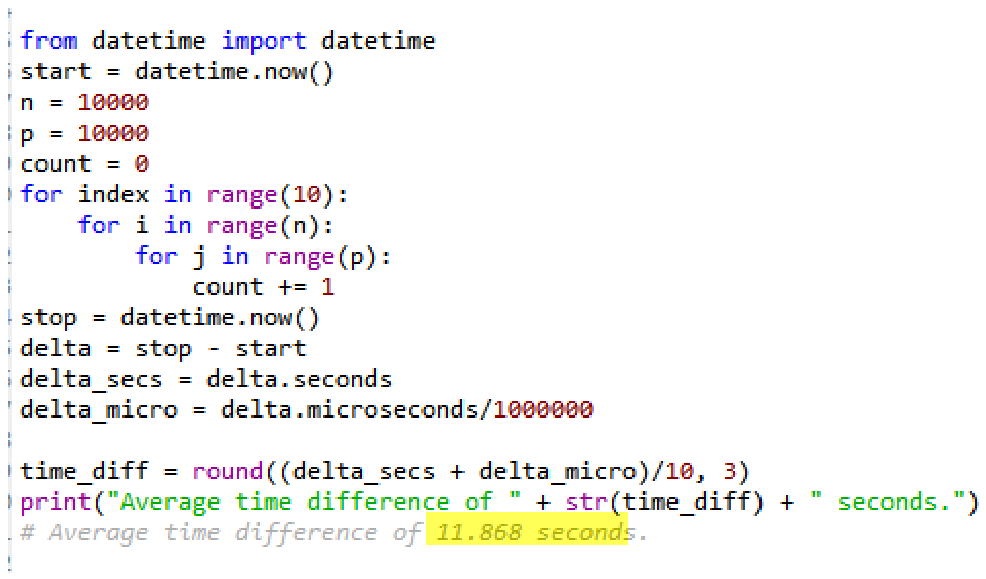
Figures 8 and 9 suggest that R handles for-loops much more efficiently than Python does.

Figure 8: R code for runtime comparison of for-loop code.



```r
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."
```

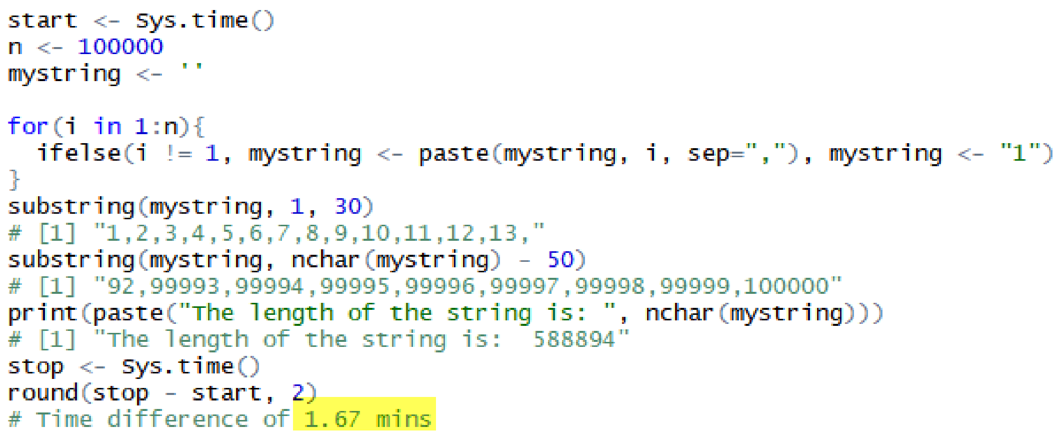
Figure 9: Python code for runtime comparison of for-loop code.



```python
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.
```

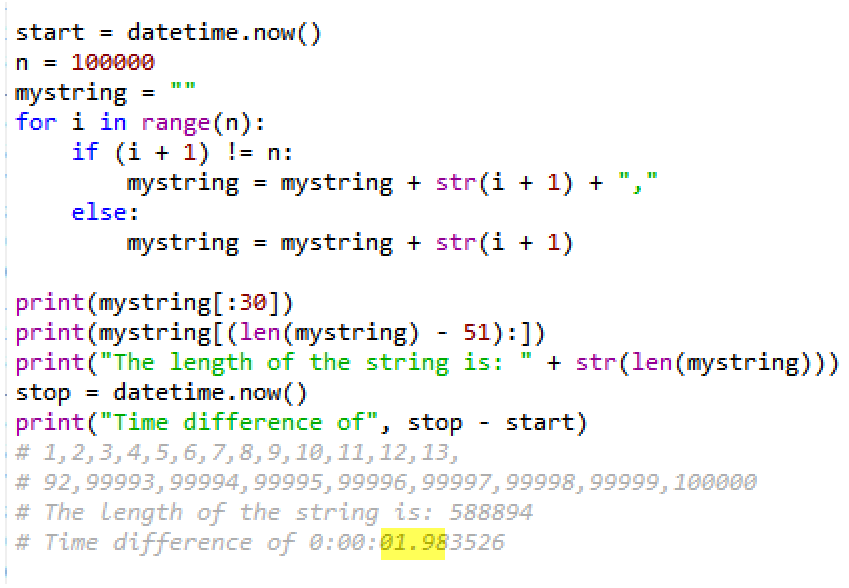
The advantage that R has with for-loops, however, is completely erased in the next test. Compare the runtime of the code in Figure 10 with the runtime of the code in Figure 11. Strings in R are character vectors, but ones in which the individual characters of the string cannot be referenced, or accessed, 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, and how they are stored in memory, makes all the difference in runtimes. (In R the character string needs contiguous memory, I believe, which forces much more copying to take place as the string grows.) 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.



```r
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.



```python
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 likely to be faster than R in certain contexts. If one is dealing with millions of records and a lot of programming is required to prepare the dataset one needs for analysis, and 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---I would seriously consider using Python. 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. The disparity is even greater if we cythonize the Python code; the Python is almost five times faster.

Still, it won't always be the case that Python is the better choice. R is an excellent tool for data analysis; I would not want to be without it. Which language is the better one to use depends on the task at hand, the package development that exists in each language, 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.