1 Introduction to R/Python Programming

There has been considerable debate over choosing R vs. Python for Data Science. I started to learn Python when I was an undergraduate in 2006. At that time I never heard of Data Science. Five years later I read an R script for the first time. In my opinion, both R and Python are great languages and are worth learning; so why not learn them together?

In this Chapter, I would give an introduction on general R and Python programming, in a parallel fashion.

1.1 Calculator

R and Python are general-purpose programming languages that can be used for writing softwares in a variety of domains. But for now, let us start from using them as basic calculators. The first thing is to have them installed. R [1] and Python [2] can be downloaded from their official website. In this book, I would keep using R 3.5 and Python 3.7.

To use R/Python as basic calculators, let's get familiar with the interactive mode. After the installation, we can type R or Python (it is case insensitive so we can also type r/python) to invoke the interactive mode. Since Python 2 is installed by default on many machines, in order to avoid invoking Python 2 we type python3.7 instead.

R

```

~ $R

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
```

[1] https://www.r-project.org
[2] https://www.python.org

```

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on—line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Python

```
~ $python3.7
Python 3.7.1 (default, Nov  6 2018, 18:45:35)
[Clang 10.0.0 (clang—1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The messages displayed by invoking the interactive mode depend on both the version of R/Python installed and the machine. Thus, you may see different messages on your local machine. As the messages said, to quit R we can type `q()`. There are 3 options prompted by asking the user if the workspace should be saved or not. Since we just want to use R as a basic calculator, we quit without saving workspace.

To quit Python, we can simply type `exit()`.

R

```
> q()
Save workspace image? [y/n/c]: n
~ $
```

Once we are inside the interactive mode, we can use R/Python as a calculator.

R

```
> 1+1
[1] 2
> 2*3+5
[1] 11
> log(2)
[1] 0.6931472
> exp(0)
```

```
[1] 1
```

Python

```
>>> 1+1
2
>>> 2*3+5
11
>>> log(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> exp(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'exp' is not defined
```

From the code snippet above, R is working as a calculator perfectly. However, errors are raised when we call `log(2)` and `exp(2)` in Python. The error messages are self-explanatory - `log` function and `exp` function don't exist in the current Python environment. In fact, `log` function and `exp` function are defined in the `math` module in Python. A module [3] is a file consisting of Python code. When we invoke the interactive mode of Python, a few built-in modules are loaded into the current environment by default. But the `math` module is not included in these built-in modules. That explains why we got the `NameError` when we try to use the functions defined in the `math` module. To resolve the issue, we should first load the functions to use by using the `import` statement as follows.

Python

```
>>> from math import log,exp
>>> log(2)
0.6931471805599453
>>> exp(0)
1.0
```

1.2 *Variable and Type*

In the previous section we have seen how to use R/Python as calculators. Now, let's see how to write real programs. First, let's define some variables.

[3] https://docs.python.org/3/tutorial/modules.html

R

```
> a=2
> b=5.0
> x='hello world'
> a
[1] 2
> b
[1] 5
> x
[1] "hello world"
> e=a*2+b
> e
[1] 9
```

Python

```
>>> a=2
>>> b=5.0
>>> x='hello world'
>>> a
2
>>> b
5.0
>>> x
'hello world'
>>> e=a*2+b
>>> e
9.0
```

Here, we defined 4 different variables `a, b, x, e`. To get the type of each variable, we can utilize the function `typeof()` in R and `type()` in Python, respectively.

R

```
> typeof(x)
[1] "character"
> typeof(e)
[1] "double"
```

Python

```
>>> type(x)
<class 'str'>
>>> type(e)
<class 'float'>
```

The type of `x` in R is called character, and in Python is called str.

1.3 *Functions*

We have seen two functions `log` and `exp` when we use R/Python as calculators. A function is a block of code which performs a specific task. A major purpose of wrapping a block of code into a function is to reuse the code.

It is simple to define functions in R/Python.

R

```
> fun1=function(x){return(x*x)}
> fun1
function(x){return(x*x)}
> fun1(2)
[1] 4
```

Python

```
>>> def fun1(x):
...    return x*x # note the indentation
...
>>> fun1(2)
4
```

Here, we defined a function `fun1` in R/Python. This function takes `x` as input and returns the square of `x`. When we call a function, we simply type the function name followed by the input argument inside a pair of parentheses. It is worth noting that input or output are not required to define a function. For example, we can define a function `fun2` to print `Hello World!` without input and output.

One major difference between R and Python codes is that Python codes are structured with indentation. Each logical line of R/Python code belongs to a certain group. In R, we use `{}` to determine the grouping of statements. However, in Python we use leading whitespace (spaces and tabs) at the beginning of a logical line to compute the indentation level of the line, which is used to determine the statements' grouping. Let's see what happens if we remove the leading whitespace in the Python function above.

Python

```
>>> def fun1(x):
... return x*x # note the indentation
  File "<stdin>", line 2
    return x*x # note the indentation
         ^
IndentationError: expected an indented block
```

We got an `IndentationError` because of missing indentation.

R

```
> fun2=function(){print('Hello
   World!')}
> fun2()
[1] "Hello World!"
```

Python

```
>>> def fun2(): print('Hello World
   !')
...
>>> fun2()
Hello World!
```

Let's go back to `fun1` and have a closer look at the `return`. In Python, if we want to return something we have to use the keyword `return` explicitly. `return` in R is a function but it is not a function in Python and that is why no parenthesis follows `return` in Python. In R, `return` is not required even though we need to return something from the function. Instead, we can just put the variables to return in the last line of the function defined in R. That being said, we can define `fun1` as follows.

R

```
> fun1=function(x){x*x}
```

1.4 *Control flows*

To implement a complex logic in R/Python, we may need control flows.

If/else

Let's define a function to return the absolute value of input.

R

```
> fun3=function(x){
+   if (x>=0){
+     return(x)}
+   else{
+     return(-x)}
+ }
> fun3(2.5)
[1] 2.5
> fun3(-2.5)
[1] 2.5
```

Python

```
>>> def fun3(x):
...   if x>=0:
...     return x
...   else:
...     return -x
...
>>> fun3(2.5)
2.5
>>> fun3(-2.5)
2.5
```

The code snippet above shows how to use `if/else` in R/Python. The subtle difference between R and Python is that the condition after `if` must be embraced by parenthesis in R but it is optional in Python.

We can also put `if` after `else`. But in Python, we use `elif` as a shortcut.

R

```
> fun4=function(x){
+   if (x==0){
+     print('zero')}
+   else if (x>0){
+     print('positive')}
+   else{
+     print('negative')}
+ }
> fun4(0)
[1] "zero"
> fun4(1)
[1] "positive"
> fun4(-1)
[1] "negative"
```

Python

```
>>> def fun4(x):
...   if x==0:
...     print('zero')
...   elif x>0:
...     print('positive')
...   else:
...     print('negative')
...
>>> fun4(0)
zero
>>> fun4(1)
positive
>>> fun4(-1)
negative
```

For loop

Similar to the usage of `if` in R, we also have to use parenthesis after the keyword `for` in R. But in Python there should be no parenthesis after `for`.

R

```
> for (i in 1:3){print(i)}
[1] 1
[1] 2
[1] 3
```

Python

```
>>> for i in range(1,4):print(i)
...
1
2
3
```

There is something more interesting than the `for loop` itself in the snippets above. In the R code, the expression `1:3` creates a vector with elements 1,2 and 3. In the Python code, we use the `range()` function for the first time. Let's have a look at the type of them.

R

```
> typeof(1:3)
[1] "integer"
```

Python

```
>>> type(range(1,4))
<class 'range'>
```

`range()` function returns a `range` type object, which represents an immutable sequence of numbers. `range()` function can take three arguments, i.e., `range(start, stop, step)`. However, `start` and `step` are both optional. It's critical to keep in mind that the `stop` argument that defines the upper limit of the sequence is exclusive. And that is why in order to loop through 1 to 3 we have to pass 4 as the `stop` argument to `range()` function. The `step` argument specifies how much to increase from one number to the next. The default values of `start` and `step` are 0 and 1, respectively.

While loop

R

```
> i=1
> while (i<=3){
+   print(i)
+   i=i+1
+ }
[1] 1
[1] 2
[1] 3
```

Python

```
>>> i=1
>>> while i<=3:
...   print(i)
...   i+=1
...
1
2
3
```

You may have noticed that in Python we can do `i+=1` to add 1 to `i`, which is not feasible in R by default. Both for loop and while loop can be nested.

Break/continue

Break/continue helps if we want to break the for/while loop earlier, or to skip a specific iteration. In R, the keyword for continue is called `next`, in contrast to `continue` in Python. The difference between `break` and `continue` is that calling `break` would exit the innermost loop (when there are nested loops, only the innermost loop is affected); while calling `continue` would just skip the current iteration and continue the loop if not finished.

R

```
> for (i in 1:3){
+   print(i)
+   if (i==1) break
+ }
[1] 1
> for (i in 1:3){
+   if (i==2){next}
+   print(i)
+ }
[1] 1
[1] 3
```

Python

```
>>> for i in range(1,4):
...   print(i)
...   if i==1: break
...
1
>>> for i in range(1,4):
...   if i==2: continue
...   print(i)
...
1
3
```

1.5 *Some built-in data structures*

In the previous sections, we haven't seen much difference between R and Python. However, regarding the built-in data structures, there are some significant differences we would see in this section.

VECTOR IN R AND LIST IN PYTHON

In R, we can use function `c()` to create a vector; A vector is a sequence of elements with the same type. In Python, we can use `[]` to create a list, which is also a sequence of elements. But the elements in a list don't need to have the same type. To get the number of elements in a vector in R, we use the function `length()`; and to get the number of elements in a list in Python, we use the function `len()`.

R

```
> x=c(1,2,5,6)
> y=c('hello','world','!')
> x
[1] 1 2 5 6
> y
[1] "hello" "world" "!"
> length(x)
[1] 4
> z=c(1,'hello')
> z
[1] "1"     "hello"
```

Python

```
>>> x=[1,2,5,6]
>>> y=['hello','world','!']
>>> x
[1, 2, 5, 6]
>>> y
['hello', 'world', '!']
>>> len(x)
4
>>> z=[1,'hello']
>>> z
[1, 'hello']
```

In the code snippet above, the first element in the variable z in R is coerced from 1 (numeric) to "1" (character) since the elements must have the same type.

To access a specific element from a vector or list, we could use []. In R, sequence types are indexed beginning with the one subscript; In contrast, sequence types in Python are indexed beginning with the zero subscript.

R

```
> x=c(1,2,5,6)
> x[1]
[1] 1
```

Python

```
>>> x=[1,2,5,6]
>>> x[1]
2
>>> x[0]
1
```

What if the index to access is out of boundary?

R

```
> x=c(1,2,5,6)
> x[-1]
[1] 2 5 6
> x[0]
numeric(0)
> x[length(x)+1]
[1] NA
> length(numeric(0))
[1] 0
> length(NA)
[1] 1
```

Python

```
>>> x=[1,2,5,6]
>>> x[-1]
6
>>> x[len(x)+1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <
      module>
IndexError: list index out of
    range
```

In Python, negative index number means indexing from the end of the list. Thus, x[-1] points to the last element and x[-2] points to the second-last element of the list. But R doesn't support indexing with negative number in the same way as Python. Specifically, in R x[-index] returns a new vector with x[index] excluded.

When we try to access with an index out of boundary, Python would throw an IndexError. The behavior of R when indexing out of boundary is more interesting. First, when we try to access x[0] in R we get a numeric(0) whose length is also 0. Since its length is 0, numeric(0) can be interpreted as an empty numeric vector. When we try to access x[length(x)+1] we get a NA. In R, there are also NaN and NULL.

NaN means "Not A Number" and it can be verified by checking its type - "double". 0/0 would result in a NaN in R. NA in R generally represents missing values. And NULL represents a NULL (empty) object. To check if a value is NA, NaN or NULL, we can use is.na(), is.nan() or is.null, respectively.

R

```
> typeof(NA)
[1] "logical"
> typeof(NaN)
[1] "double"
> typeof(NULL)
[1] "NULL"
> is.na(NA)
[1] TRUE
> is.null(NULL)
[1] TRUE
> is.nan(NaN)
```

Python

```
>>> type(None)
<class 'NoneType'>
>>> None is None
True
>>> 1 == None
False
```

In Python, there is no built-in `NA` or `NaN`. The counterpart of `NULL` in Python is `None`. In Python, we can use the `is` keyword or == to check if a value is equal to `None`.

From the code snippet above, we also notice that in R the boolean type value is written as "TRUE/FALSE", compared with "True/False" in Python. Although in R "TRUE/FALSE" can also be abbreviated as "T/F", I don't recommend to use the abbreviation.

There is one interesting fact that we can't add a `NULL` to a vector in R, but it is feasible to add a `None` to a list in Python.

R

```
> x=c(1,NA,NaN,NULL)
> x
[1]   1  NA NaN
> length(x)
[1] 3
```

Python

```
>>> x=[1,None]
>>> x
[1, None]
>>> len(x)
2
```

Beside accessing a specific element from a vector/list, we may also need to do slicing, i.e., to select a subset of the vector/list. There are two basic approaches of slicing:

- Integer-based

R

```
> x=c(1,2,3,4,5,6)
> x[2:4]
[1] 2 3 4
> x[c(1,2,5)] # a vector of indices
[1] 1 2 5
```

```
> x[seq(1,5,2)] # seq creates a vector to be used as indices
[1] 1 3 5
```

Python

```
>>> x=[1,2,3,4,5,6]
>>> x[1:4] # x[start:end] start is inclusive but end is exclusive
[2, 3, 4]
>>> x[0:5:2] # x[start:end:step]
[1, 3, 5]
```

The code snippet above uses hash character # for comments in both R and Python. Everything after # on the same line would be treated as comment (not executable). In the R code, we also used the function seq() to create a vector. When I see a function that I haven't seen before, I might either google it or use the builtin helper mechanism. Specifically, in R use ? and in Python use help().

R

```
> ?seq
```

Python

```
>>> help(print)
```

- Condition-based

  Condition-based slicing means to select a subset of the elements which satisfy certain conditions. In R, it is quite straightforward by using a boolean vector whose length is the same as the vector to slice.

R

```
> x=c(1,2,5,5,6,6)
> x[x %% 2==1] # %% is the modulo operator in R; we select the odd elements
[1] 1 5 5
> x %% 2==1 # results in a boolean vector with the same length as x
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE
```

The condition-based slicing in Python is quite different from that in R. The prerequisite is list comprehension which provides a concise way to create new lists in Python. For example, let's create a list of squares of another list.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e**2 for e in x] # ** is the exponent operator, i.e., x**y means x to
    the power of y
[1, 4, 25, 25, 36, 36]
```

We can also use if statement with list comprehension to filter a list to achieve list slicing.

Python

```
>>> x=[1,2,5,5,6,6]
>>> [e for e in x if e%2==1] # % is the modulo operator in Python
[1, 5, 5]
```

It is also common to use `if`/`else` with list comprehension to achieve more complex operations. For example, given a list x, let's create a new list y so that the non-negative elements in x are squared and the negative elements are replaced by 0s.

Python

```
>>> x=[1,-1,0,2,5,-3]
>>> [e**2 if e>=0 else 0 for e in x]
[1, 0, 0, 4, 25, 0]
```

The example above shows the power of list comprehension. To use `if` with list comprehension, the `if` statement should be placed in the end after the `for` loop statement; but to use `if`/`else` with list comprehension, the `if`/`else` statement should be placed before the `for` loop statement.

We can also modify the value of an element in a vector/list variable.

R

```
> x=c(1,2,3)
> x[1]=-1
> x
[1] -1  2  3
```

Python

```
>>> x=[1,2,3]
>>> x[0]=-1
>>> x
[-1, 2, 3]
```

Although the vector structure in R and the list structure in Python looks similar regarding their usages and purposes, the implementation of these two structures are essentially different. The list structure in Python is mutable. A mutable object can be changed after it is created, but an immutable object can't. However, the mutability of vector in R is a bit of complicated. If we change the value of an element in a vector without changing the type of the element, the vector is mutable. If we change an element to another type, the behavior of the vector is immutable. A variable itself is a reference or pointer to an object (usually stored in the machine's memory). To check the mutability of a variable, we can trace the memory address.

R

```
> x=c(1:3)
> tracemem(x) # print the memory address of x whenever the address changes
[1] "<0x7ff360c95c08>"
> x[1]=-x[1] # type not changed, i.e., from integer to integer
> tracemem(x)
[1] "<0x7ff360c95c08>"
> x[1]=-1.0
tracemem[0x7ff360c95c08 -> 0x7ff3604692d8]:
```

Python

```
>>> x=list(range(1,1001)) # list() convert a range object to a list
>>> hex(id(x)) # print the memory address of x
'0x10592d908'
>>> x[0]=1.0 # from integer to float
>>> hex(id(x))
'0x10592d908'
```

From the code snippet above, in Python the memory address doesn't change after we change the value of the first element because list in Python is mutable. When we try to modify the value of x[1] in R, the memory address of x doesn't change. (you probably would see different addresses on your machine). But when we change the value of x[1] from the integer type to a double type, the memory address got changed. It's worth noting since R 3.5 arithmetic sequences created by 1:n, seq_along, and the like now use compact internal representations via the ALTREP framework [4]. Let's the example below.

R

```
> x=1:3
> tracemem(x)
[1] "<0x7f828e84c110>"
> .Internal(inspect(x))
@7f828e84c110 13 INTSXP g0c0 [NAM(3),TR]  1 : 3 (compact)
> x[1]=2L
tracemem[0x7f828e84c110 -> 0x7f828fe49848]:
```

R

```
> x=c(1:3)
> tracemem(x)
[1] "<0x7f828fe498c8>"
> .Internal(inspect(x))
@7f828fe498c8 13 INTSXP g0c2 [NAM(1),TR] (len=3, tl=0) 1,2,3
> x[1]=2L
> tracemem(x)
[1] "<0x7f828fe498c8>"
```

[4] https://cran.r-project.org/doc/manuals/r-devel/NEWS.html

Two or multiple vectors/lists can be concatenated easily.

R

```
> x=c(1,2)
> y=c(3,4)
> z=c(5,6,7,8)
> c(x,y,z)
[1] 1 2 3 4 5 6 7 8
```

Python

```
>>> x=[1,2]
>>> y=[3,4]
>>> z=[5,6,7,8]
>>> x+y+z
[1, 2, 3, 4, 5, 6, 7, 8]
```

As the list structure in Python is mutable, there are many things we can do with list.

Python

```
>>> x=[1,2,3]
>>> x.append(4) # append a single value to the list x
>>> x
[1, 2, 3, 4]
>>> y=[5,6]
>>> x.extend(y) # extend list y to x
>>> x
[1, 2, 3, 4, 5, 6]
>>> last=x.pop() # pop the last elememt from x
>>> last
6
>>> x
[1, 2, 3, 4, 5]
```

Is there any immutable data structure in Python? Yes, for example tuple is immutable, which contains a sequence of elements. The element accessing and subset slicing of tuple is following the same rules of list in Python.

Python

```
>>> x=(1,2,3,) # use () to create a tuple in Python, it is better to always
    put a comma in the end
>>> type(x)
<class 'tuple'>
>>> len(x)
3
>>> x[0]
1
>>> x[0]=-1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

I like the list structure in Python much more than the vector structure in R. list in Python has a lot more useful features which can be found from the python official documentation [5].

Array

Array is one of the most important data structures in scientific programming. In R, there is also an object type "matrix", but according to my own experience, we can almost ignore its existence and use array instead. We can definitely use list as array in Python, but lots of linear algebra operations are not supported for the list type. Fortunately, there is a Python package `numpy` off the shelf.

R

```
> x=1:12
> array1=array(x,c(4,3)) # convert vector x to a 4 rows * 3 cols array
> array1
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> y=1:6
> array2=array(y,c(3,2)) # convert vector y to a 3 rows * 2 cols array
> array2
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> array3 = array1 %*% array2 # %*% is the matrix multiplication operator
> array3
     [,1] [,2]
[1,]   38   83
[2,]   44   98
[3,]   50  113
[4,]   56  128
> dim(array3) # get the dimension of array3
[1] 4 2
```

Python

```
>>> import numpy as np # we import the numpy module and alias it as np
```

[5] https://docs.python.org/3/tutorial/datastructures.html

```
>>> array1=np.reshape(list(range(1,13)),(4,3)) # convert a list to a 2d np.
    array
>>> array1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> type(array1)
<class 'numpy.ndarray'>
>>> array2=np.reshape(list(range(1,7)),(3,2))
>>> array2
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> array3=np.dot(array1,array2) # matrix multiplication using np.dot()
>>> array3
array([[ 22,  28],
       [ 49,  64],
       [ 76, 100],
       [103, 136]])
>>> array3.shape # get the shape(dimension) of array3
(4, 2)
```

You may have noticed that the results of the R code snippet and Python code snippet are different. The reason is that in R the conversion from a vector to an array is by-column; but in numpy the reshape from a list to an 2D numpy.array is by-row. There are two ways to reshape a list to a 2D numpy.array by column.

Python

```
>>> array1=np.reshape(list(range(1,13)),(4,3),order='F') # use order='F'
>>> array1
array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]])
>>> array2=np.reshape(list(range(1,7)),(2,3)).T # use .T to transpose an array
>>> array2
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.dot(array1,array2) # now we get the same result as using R
array([[ 38,  83],
       [ 44,  98],
       [ 50, 113],
       [ 56, 128]])
```

To learn more about numpy, the official website [6]. has great documentation/tutorials.

LIST IN R AND DICTIONARY IN PYTHON

Yes, in R there is also an object type called list. The major difference between a vector and a list in R is that a list could contain different types of elements. list in R supports integer-based accessing using [[]] (compared to [] for vector).

R

```
> x=list(1,'hello world!')
> x
[[1]]
[1] 1

[[2]]
[1] "hello world!"

> x[[1]]
[1] 1
> x[[2]]
[1] "hello world!"
> length(x)
[1] 2
```

The mutability of the list structure in R is similar to the vector structure. The difference is that when we change the type of an element in a list, the memory address doesn't change in general.

R

```
> x=list(c(1:3),'Hello World!')
> tracemem(x)
[1] "<0x7f828fe497c8>"
> x[[1]]=1.0
> x[[2]]=2.0
> x
[[1]]
[1] 1

[[2]]
[1] 2

```

[6] http://www.numpy.org

```
> tracemem(x)
[1] "<0x7f828fe497c8>"
```

list in R could be named and support accessing by name via either `[[]]` or `$` operator. But vector in R can also be named and support accessing by name.

R

```
> x=c('a'=1,'b'=2)
> names(x)
[1] "a" "b"
> x['b']
b
2
> l=list('a'=1,'b'=2)
> l[['b']]
[1] 2
> l$b
[1] 2
> names(l)
[1] "a" "b"
```

However, elements in list in Python can't be named as R. If we need the feature of accessing by name in Python, we can use the dictionary structure. If you used Java before, you may consider dictionary in Python as the counterpart of HashMap in Java. Essentially, a dictionary in Python is a collection of key:value pairs.

Python

```
>>> x={'a':1,'b':2} # {key:value} pairs
>>> x
{'a': 1, 'b': 2}
>>> x['a']
1
>>> x['b']
2
>>> len(x) # number of key:value pairs
2
>>> x.pop('a') # remove the key 'a' and we get its value 1
1
>>> x
{'b': 2}
```

Unlike dictionary in Python, list in R doesn't support the `pop()` operation. Thus, in order to modify a list in R, a new one would be created explicitly or implicitly.

DATA.FRAME, DATA.TABLE AND PANDAS

data.frame is a built-in type in R for data manipulation. data.table [7] is an R package and pandas [8] is a Python module. To use data.table and pandas, we have to install the them. Although data.frame is a built-in type, it is not quite efficient for many operations. I would suggest to use data.table whenever possible. dplyr [9] is also a very popular package in R for data manipulation. But I favor data.table over dplyr. Many good online resources are available online to learn data.table and pandas. Thus, I would not cover the usage of these tools for now.

Object-oriented programming (OOP) in R/Python

All the codes we wrote above follow the procedural programming paradigm [10]. We can also do functional programming (FP) and OOP in R/Python. In this section, let's focus on OOP in R/Python.

Class is the key concept in OOP. In R there are two commonly used built-in systems to define classes, i.e., S3 and S4. In addition, there is an external package R6 [11] which defines R6 classes. S3 is a light-weight system but its style is quite different from OOP in many other programming languages. S4 system follows the principles of modern object oriented programming much better than S3. However, the usage of S4 classes is quite tedious. I would ignore S3/S4 and introduce R6, which is more close to the class in Python.

Let's build a class in R/Python to represent complex numbers.

R

```
> library(R6) # load the R6 package
>
> Complex = R6Class("Complex",
+ public = list( # only elements declared in this list are accessible by the
  object of this class
+ real = NULL,
+ imag = NULL,
+ # the initialize function would be called automatically when we create an
  object of the class
+ initialize = function(real,imag){
+     # call functions to change real and imag values
+   self$set_real(real)
+   self$set_imag(imag)
+ },
+ # define a function to change the real value
+ set_real = function(real){
+   self$real=real
+ },
+ # define a function to change the imag value
+ set_imag = function(imag){
```

[7] https://cran.r-project.org/web/packages/data.table/index.html
[8] https://pandas.pydata.org/
[9] https://dplyr.tidyverse.org/
[10] https://en.wikipedia.org/wiki/Comparison_of_programming_paradigms
[11] https://cran.r-project.org/web/packages/R6/index.html

```
+     self$imag=imag
+ },
+ # override print function
+ print = function(){
+     cat(paste0(as.character(self$real),'+',as.character(self$imag),'j'),'\n')
+ }
+ )
+ )
> # let's create a complex number object based on the Complex class we defined
    above using the new function
> x = Complex$new(1,2)
> x
1+2j
> x$real # the public attributes of x could be accessed by $ operator
[1] 1
```

Python

```
>>> class Complex:
...     # the __init__ function would be called automatically when we create an
    object of the class
...     def __init__(self,real,imag):
...         self.real = None
...         self.imag = None
...         self.set_real(real)
...         self.set_imag(imag)
...     # define a function to change the real value
...     def set_real(self,real):
...         self.real=real
...     # define a function to change the imag value
...     def set_imag(self,imag):
...         self.imag=imag
...     def __repr__(self):
...         return "{0}+{1}j".format(self.real,self.imag)
...
>>> x = Complex(1,2)
>>> x
1+2j
>>> x.real # different from the $ operator in R, here we use . to access the
    attribute of an object
1
```

By overriding the print function in the R6 class, we can have the object printed in the format of `real+imag j`. To achieve the same effect in Python, we override the method `__repr__`. In Python, we call the functions defined in classes as methods. And overriding a method means changing the implementation of a method provided by one of its ancestors. To understand the concept of ancestors in OOP, one needs to understand the concept of inheritance [12].

You may be curious of the double underscore surrounding the methods, such as `__init__` and `__repr__`. These methods are well-known as magic methods [13]. Magic methods could be very handy if we use them in the suitable cases. For example, we can use the magic method `__add__` to implement the + operator for the `Complex` class we defined above.

In the definition of the magic method `__repr__` in the Python code, the `format` method of `str` object [14] is used.

Python

```
>>> class Complex:
...   def __init__(self,real,imag):
...     self.real = None
...     self.imag = None
...     self.set_real(real)
...     self.set_imag(imag)
...   def set_real(self,real):
...     self.real=real
...   def set_imag(self,imag):
...     self.imag=imag
...   def __repr__(self):
...     return "{0}+{1}j".format(self.real,self.imag)
...   def __add__(self,another):
...     return Complex(self.real+another.real,self.imag+another.imag)
...
>>> x=Complex(1,2)
>>> y=Complex(2,4)
>>> x+y # + operator works now
3+6j
```

We can also implement the + operator for `Complex` class in R like what we have done for Python.

R

```
>  `+.Complex` = function(x,y){
+   Complex$new(x$real+y$real,x$imag+y$imag)
+ }
> x=Complex$new(1,2)
> y=Complex$new(2,4)
```

[12] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[13] https://rszalski.github.io/magicmethods/
[14] https://docs.python.org/3.7/library/string.html

```
> x+y
3+6j
```

The most interesting part of the code above is `` `+.Complex` ``. First, why do we use ``` `` ``` to quote the function name? Before getting into this question, let's have a look at the Python 3's variable naming rules [15].

```
Within the ASCII range (U+0001..U+007F), the valid characters for identifiers
    (also referred to as names) are the same as in Python 2.x: the uppercase
    and lowercase letters A through Z, the underscore _ and, except for the
    first character, the digits 0 through 9.
```

According to the rule, we can't declare a variable with name `2x`. Compared with Python, in R we can also use `.` in the variable names [16]. However, there is a workaround to use invalid variable names in R with the help of ``` `` ```.

R

```
> 2x = 5
Error: unexpected symbol in "2x"
> .x = 3
> .x
[1] 3
> `+2x%` = 0
> `+2x%`
[1] 0
```

Python

```
>>> 2x = 5
  File "<stdin>", line 1
    2x = 5
     ^
SyntaxError: invalid syntax
>>> .x = 3
  File "<stdin>", line 1
    .x = 3
    ^
SyntaxError: invalid syntax
```

Now it is clear the usage of ``` `` ``` in `` `+.Complex` `` is to define a function with invalid name. Placing `.Complex` after + is related to S3 method dispatching which would not be discussed here.

1.6 *Miscellaneous*

There are some items that I haven't discussed so far, which are also important in order to master R/Python.

Package/module installation

- Use `install.packages()` function in R
- Use R IDE to install packages
- Use `pip` [17] to install modules in Python

[15] https://docs.python.org/3.3/reference/lexical_analysis.html
[16] https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Identifiers

[17] https://packaging.python.org/tutorials/installing-packages/

Virtual Environment

Virtual environment is a tool to mange dependencies in Python. There are different ways to create virtual environments in Python. But I suggest to use the `venv` module shipped with Python 3. Unfortunately, there is nothing like a real virtual environment in R as far as I know although there quite a few of packages management tools/packages.

<- vs. =

If you have known R before, you probably heard of the advice [18] to use <− to rather than = for value assignment. I always use = for value assignment. Let's see an example when <− makes a difference when we do value assignment.

R

```
> x=1
> a=list(x <- 2)
> a
[[1]]
[1] 2

> x
[1] 2
```

R

```
> x=1
> a=list(x = 2)
> a
$x
[1] 2

> x
[1] 1
```

[18] https://google.github.io/styleguide/Rguide.xml

2
More on R/Python Programming

We have learned quite a few of basic R/Python programming in the previous chapter. I hope this chapter could be used as an intermediate level R/Python programming tutorial. There are a few topics to cover, including debugging, vectorization and some other useful features of R/Python.

2.1 Write/run R/Python scripts

In chapter 1 we are coding within the interactive mode of R/Python. When we are working on the real world projects, using an Integrated development environment (IDE) is a more pragmatic choice. There are not many choices for R IDE, and among them RStudio [19] is the best one I have used so far. As for Python, I would recommend either Visual Studio Code [20] or PyCharm [21]. But of course, you could use any text editor to write R/Python scripts.

Let's write a simple script to print `Hello World!` in R/Python. I have made a directory `chapter2` on my disk, the R script is saved as `hello_world.R` and the Python script is saved as `hello_world.py`, inside the directory.

R

chapter2/hello_world.R

```r
print("Hello World!")
```

Python

chapter2/hello_world.py

```python
print("Hello World!")
```

There are a few ways to run the R script. For example, we can run the script from the console with the `r -f filename` command. Also, we can open the R interactive session and use the `source()` function. I would recommend the second approach with `source()` function. As for the Python script, we can run it from the console.

[19] https://www.rstudio.com/products/rstudio/
[20] https://www.jetbrains.com/pycharm/
[21] https://www.jetbrains.com/pycharm/

R

```
chapter2 $ls
hello_world.R hello_world.py
chapter2 $r -f hello_world.R
> print("Hello World!")
[1] "Hello World!"

chapter2 $r
> source('hello_world.R')
[1] "Hello World!"
```

Python

```
chapter2 $ls
hello_world.R hello_world.py

chapter2 $python3.7 hello_world.py
Hello World!
```

2.2 *Debugging in R/Python*

Debugging is one of the most important aspects of programming. What is debugging in programming? The programs we write might include errors/bugs and debugging is a step-by-step process to find and remove the errors/bugs in order to get the desired results.

If you are smart enough or the bugs are evident enough then you can debug the program on your mind without using a computer at all. But in general we need some tools/techniques to help us with debugging.

Print

Most of programming languages provide the functionality of printing, which is a natural way of debugging. By trying to place `print` statements at different positions we may finally catch the bugs. When I use `print` to debug, it's feeling like playing the game of minesweeper. In Python, there is a module called `logging` [22] which could be used for debugging like the `print` function, but in a more elegant fashion.

Browser in R and pdb in Python

In R, there is a function `browser()` which interrupts the execution and allows the inspection of the current environment. Similarly, there is a module `pdb` in Python that provides more debugging features. We would only focus on the basic usages of `browser()` and the `set_trace()` function in `pdb` module. The essential difference between debugging using `print()` and `browser()` and `set_trace()` is that the latter functions allows us to debug in an interactive mode.

Let's write a function which takes a sorted vector/list `v` and a target value `x` as input and returns the leftmost index `pos` of the sorted vector/list so that `v[pos]>=x`. Since `v` is already sorted, we may simply loop through it from left to right to find `pos`.

[22] https://docs.python.org/3/library/logging.html

R

chapter2/find_pos.R

```r
find_pos=function(v,x){
  for (i in 1:length(v)){
    if (v[i]>=x){
      return(i)
    }
  }
}

v=c(1,2,5,10)
print(find_pos(v,-1))
print(find_pos(v,4))
print(find_pos(v,11))
```

Python

chapter2/find_pos.py

```python
def find_pos(v,x):
  for i in range(len(v)):
    if v[i]>=x:
      return i

v=[1,2,5,10]
print(find_pos(v,-1))
print(find_pos(v,4))
print(find_pos(v,11))
```

Now let's run these two scripts.

R

```
chapter2 $r
> source('find_pos.R')
[1] 1
[1] 3
NULL
```

Python

```
chapter2 $python3.7 find_pos.py
0
2
None
```

When `x=11`, the function returns `NULL` in R and `None` in Python because there is no such element in v larger than `x`. The implementation above is trivial, but not efficient. If you have some background in data structures and algorithms, you probably know this question can be solved by binary search. The essential idea of binary search comes from Divide-and-conquer [23]. Since v is already sorted, we may divide it into two partitions by cutting it from the middle, and then we get the left partition and the right partition. v is sorted implies that both the left partition and the right partition are also sorted. If the target value `x` is larger than the rightmost element in the left partition, we can just discard the left partition and search `x` within the right partition. Otherwise, we can discard the right partition and search `x` within the left partition. Once we have determined which partition to search, we may apply the idea recursively so that in each step we reduce the size of v by half. If the length of v is denoted as n, in terms of big O notation [24], the run time complexity of binary search is $\mathcal{O}(\log n)$, compared with $\mathcal{O}(n)$ of the for-loop implementation.

The code below implements the binary search solution to our question (It is more intuitive to do it with recursion but here I write it with iteration since tail recursion optimization [25] in R/Python is not supported).

[23] https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm
[24] https://en.wikipedia.org/wiki/Big_O_notation
[25] https://en.wikipedia.org/wiki/Tail_call

R

chapter2/find_binary_search_buggy.R

```r
binary_search_buggy=function(v,x){
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end) %/% 2 # %/% is the floor division operator
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
  return(start)
}
v=c(1,2,5,10)
print(binary_search_buggy(v,-1))
print(binary_search_buggy(v,5))
print(binary_search_buggy(v,11))
```

Python

chapter2/find_binary_search_buggy.py

```python
def binary_search_buggy(v,x):
  start,end = 0,len(v)-1
  while start<end:
    mid = (start+end)//2  # // is the floor division operator
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start

v=[1,2,5,10]
print(binary_search_buggy(v,-1))
print(binary_search_buggy(v,5))
print(binary_search_buggy(v,11))
```

Now let's run these two binary_search scripts.

R

```
chapter2 $r
> source('binary_search_buggy.R')
[1] 1
[1] 3
[1] 4
```

Python

```
chapter2 $python3.7
    binary_search_buggy.py
0
2
3
```

The binary search solutions don't work as expected when x=11. We write two new scripts.

R

chapter2/find_binary_search_buggy_debug.R

```
binary_search_buggy=function(v,x){
  browser()
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end)
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
  return(start)
}
v=c(1,2,5,10)
print(binary_search_buggy(v,11))
```

Python

chapter2/find_binary_search_buggy_debug.py

```
from pdb import set_trace
def binary_search_buggy(v,x):
  set_trace()
  start,end = 0,len(v)−1
  while start<end:
    mid = (start+end)//2
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start

v=[1,2,5,10]
print(binary_search_buggy(v,11))
```

Let's try to debug the programs with the help of browser() and set_trace().

R

```
> source('binary_search_buggy_debug.R')
Called from: binary_search_buggy(v, 11)
Browse[1]> ls()
[1] "v" "x"
Browse[1]> n
debug at binary_search_buggy_debug.R#3: start = 1
Browse[2]> n
debug at binary_search_buggy_debug.R#4: end = length(v)
Browse[2]> n
debug at binary_search_buggy_debug.R#5: while (start < end) {
    mid = (start + end)%/%2
    if (v[mid] >= x) {
        end = mid
    }
    else {
        start = mid + 1
    }
}
Browse[2]> n
debug at binary_search_buggy_debug.R#6: mid = (start + end)%/%2
Browse[2]> n
debug at binary_search_buggy_debug.R#7: if (v[mid] >= x) {
    end = mid
} else {
    start = mid + 1
}
Browse[2]> n
debug at binary_search_buggy_debug.R#10: start = mid + 1
Browse[2]> n
debug at binary_search_buggy_debug.R#5: (while) start < end
Browse[2]> n
debug at binary_search_buggy_debug.R#6: mid = (start + end)%/%2
Browse[2]> n
```

```
debug at binary_search_buggy_debug.R#7: if (v[mid] >= x) {
    end = mid
} else {
    start = mid + 1
}
Browse[2]> n
debug at binary_search_buggy_debug.R#10: start = mid + 1
Browse[2]> n
debug at binary_search_buggy_debug.R#5: (while) start < end
Browse[2]> start
[1] 4
Browse[2]> n
debug at binary_search_buggy_debug.R#13: return(start)
Browse[2]> n
[1] 4
```

In the R code snippet above, we placed the `browser()` function on the top of the function `binary_search_buggy`. Then when we call the function we enter into the debugging environment. By calling `ls()` we see all variables in the current debugging scope, i.e., `v,x`. Typing `n` will evaluate the next statement. After typing `n` a few times, we finally exit from the while loop because `start = 4` such that `start < end` is FALSE. As a result, the function just returns the value of start, i.e., 4. To exit from the debugging environment, we can type `Q`; to continue the execution we can type `c`.

The root cause is that we didn't deal with the corner case when the target value `x` is larger than the last/largest element in v correctly.

Let's debug the Python function using `pdb` module.

Python

```
chapter2 $python3.7 binary_search_buggy_debug.py
> chapter2/binary_search_buggy_debug.py(4)binary_search_buggy()
-> start,end = 0,len(v)-1
(Pdb) n
> chapter2/binary_search_buggy_debug.py(5)binary_search_buggy()
-> while start<end:
(Pdb) l
  1   from pdb import set_trace
  2   def binary_search_buggy(v,x):
  3     set_trace()
  4     start,end = 0,len(v)-1
  5  ->   while start<end:
  6       mid = (start+end)//2
  7       if v[mid]>=x:
  8         end = mid
  9       else:
```

```
 10         start = mid+1
 11     return start
(Pdb) b 7
Breakpoint 1 at chapter2/binary_search_buggy_debug.py:7
(Pdb) c
> chapter2/binary_search_buggy_debug.py(7)binary_search_buggy()
-> if v[mid]>=x:
(Pdb) c
> chapter2/binary_search_buggy_debug.py(7)binary_search_buggy()
-> if v[mid]>=x:
(Pdb) mid
2
(Pdb) n
> chapter2/binary_search_buggy_debug.py(10)binary_search_buggy()
-> start = mid+1
(Pdb) n
> chapter2/binary_search_buggy_debug.py(5)binary_search_buggy()
-> while start<end:
(Pdb) start
3
(Pdb) n
> chapter2/binary_search_buggy_debug.py(11)binary_search_buggy()
-> return start
```

Similar to R, command `n` would evaluate the next statement in `pdb`. Typing command `l` would show the current line of current execution. Command `b line_number` would set the corresponding line as a break point; and `c` would continue the execution until the next breakpoint (if exists).

In R, besides the `browser()` function there are a pair of functions `debug()`and `undebug()` which are also very handy when we try to debug a function; especially when the function is wrapped in a package. More specifically, the `debug` function would invoke the debugging environment whenever we call the function to debug. See the example below how we invoke the debugging environment for the `sd` function (standard deviation calculation).

R

```
> x=c(1,1,2)
> debug(sd)
> sd(x)
debugging in: sd(x)
debug: sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
    na.rm = na.rm))
Browse[2]> ls()
[1] "na.rm" "x"
Browse[2]> Q
```

```
> undebug(sd)
> sd(x)
[1] 0.5773503
```

The binary_search solutions are fixed below.

R

chapter2/find_binary_search.py

```
binary_search=function(v,x){
  if (x>v[length(v)]){return(NULL)}
  start = 1
  end = length(v)
  while (start<end){
    mid = (start+end)
    if (v[mid]>=x){
      end = mid
    }else{
      start = mid+1
    }
  }
  return(start)
}
```

Python

chapter2/find_binary_search.py

```
def binary_search(v,x):
  if x>v[-1]: return
  start,end = 0,len(v)-1
  while start<end:
    mid = (start+end)//2
    if v[mid]>=x:
      end = mid
    else:
      start = mid+1
  return start
```

2.3 *Benchmarking*

By benchmarking, I mean measuring the entire operation time of a piece of program. There is another term called profiling which is related to benchmarking. But profiling is more complex since it commonly aims at understanding the behavior of the program and optimizing the program in terms of time elapsed during the operation.

In R, I like using the `microbenchmark` package. And in Python, `timeit` module is a good tool to use when we want to benchmark a small bits of Python code.

As mentioned before, the run time complexity of binary search is better than that of a for-loop search. We can do benchmarking to compare the two algorithms.

R

chapter2/benchmark.R

```r
library(microbenchmark)
source('binary_search.R')
source('find_pos.R')

v=1:10000

# call each function 1000 times;
# each time we randomly select an integer as the target value

# for—loop solution
set.seed(2019)
print(microbenchmark(find_pos(v,sample(10000,1)),times=1000))
# binary—search solution
set.seed(2019)
print(microbenchmark(binary_search(v,sample(10000,1)),times=1000))
```

In the R code above, `times=1000` means we want to call the function `1000` times in the benchmarking process. The `sample()` function is used to draw samples from a set of elements. Specifically, we pass the argument 1 to `sample()` to draw a single element. It's the first time we use `set.seed()` function in this book. In R/Python, we draw random numbers based on the pseudorandom number generator (PRNG) algorithm [26]. The sequence of numbers generated by PRNG is completed determined by an initial value, i.e., the seed. Whenever a program involves the usage of PRNG, it is better to set the seed in order to get replicable results (see the example below).

R

```r
> set.seed(2019)
> rnorm(1)
[1] 0.7385227
> rnorm(1)
[1] —0.5147605
```

R

```r
> set.seed(2019)
> rnorm(1)
[1] 0.7385227
> set.seed(2019)
> rnorm(1)
[1] 0.7385227
```

[26] https://en.wikipedia.org/wiki/Pseudorandom_number_generator

Now let's run the R script to see the benchmarking result. llr

```
> source('benchmark.R')
Unit: microseconds
                              expr  min        lq      mean    median        uq
                                 max
 find_pos(v, sample(10000, 1)) 3.96 109.5385 207.6627 207.5565 307.8875
    536.171
 neval
  1000
Unit: microseconds
                                   expr   min      lq     mean median      uq      max
 binary_search(v, sample(10000, 1)) 5.898 6.3325 14.2159 6.6115 7.3635 6435.57
 neval
  1000
```

The binary_search solution is much more efficient based on the benchmarking result. Doing the same benchmarking in Python is a bit of complicated.

Python

chapter2/benchmark.py

```python
from binary_search import binary_search
from find_pos import find_pos
import timeit
import random

v=list(range(1,10001))

def test_for_loop(n):
  random.seed(2019)
  for _ in range(n):
    find_pos(v,random.randint(1,10000))

def test_bs(n):
  random.seed(2019)
  for _ in range(n):
    binary_search(v,random.randint(1,10000))

# for-loop solution
print(timeit.timeit('test_for_loop(1000)',setup='from __main__ import
    test_for_loop',number=1))
# binary_search solution
print(timeit.timeit('test_bs(1000)',setup='from __main__ import test_bs',
    number=1))
```

The most interesting part of the Python code above is `from __main__ import`. Let's ignore it for now, and we would revisit it later.

Below is the benchmarking result in Python (the unit is second).

Python

```
chapter2 $python3 benchmark.py
0.284618441
0.00396658900000002
```

2.4 *Vectorization*

In parallel computing, automatic vectorization [27] means a program in a scalar implementation is converted to a vector implementation which process multiple pairs of operands simultaneously by compilers that feature auto-vectorization. For example, let's calculate the element-wise sum of two arrays `x` and `y` of the same length in C programming language.

```
int x[4] = {1,2,3,4};
int y[4] = {0,1,2,3};
int z[4];
for (int i=0;i<4;i++){
    z[i]=x[i]+y[i];
}
```

The C code above might be vectorized by the compiler so that the actual number of iterations performed could be less than 4. If 4 pairs of operands are processed at once, there would be only 1 iteration. Automatic vectorization may make the program runs much faster in some languages like C. However, when we talk about vectorization in R/Python, it is different from automatic vectorization. Vectorization in R/Python usually refers to the human effort paid to avoid for-loops. First, let's see some examples of how for-loops may slow your programs in R/Python.

R

chapter2/vectorization_1.R

```
library(microbenchmark)

# generate n standard normal r.v
rnorm_loop = function(n){
x=rep(0,n)
for (i in 1:n) {x[i]=rnorm(1)}
}

rnorm_vec = function(n){
```

[27] https://en.wikipedia.org/wiki/Automatic_vectorization

```
x=rnorm(n)
}

n=100
# for loop
print(microbenchmark(rnorm_loop(n),times=1000))
# vectorize
print(microbenchmark(rnorm_vec(n),times=1000))
```

Running the R code results in the following result on my local machine.
llr

```
> source('vectorization_1.R')
Unit: microseconds
          expr     min      lq     mean   median      uq     max neval
 rnorm_loop(n) 131.622 142.699 248.7603 145.3995 270.212 16355.6  1000
Unit: microseconds
         expr   min    lq     mean median    uq      max neval
 rnorm_vec(n) 6.696 7.128 10.87463  7.515 8.291 2422.338  1000
```

Python

```python
import timeit
import numpy as np

def rnorm_for_loop(n):
  x=[0]*n # create a list with n 0s
  np.random.seed(2019)
  for _ in range(n):
    np.random.normal(0,1,1)

def rnorm_vec(n):
  np.random.seed(2019)
  x = np.random.normal(0,1,n)

print("for loop")
print(f'{timeit.timeit("rnorm_for_loop(100)",setup="from __main__ import
    rnorm_for_loop",number=1000):.6f} seconds')
print("vectorized")
print(f'{timeit.timeit("rnorm_vec(100)",setup="from __main__ import rnorm_vec
    ",number=1000):.6f} seconds')
```

Please note that in this Python example we are using the `random` submodule of `numpy` module instead of the built-in `random` module since `random` module doesn't provide the vectorized version of random number generation function. Running the Python code results in the following result on my local machine.

Python

```
chapter2 $python3.7 vectorization_1.py
for loop
0.258466 seconds
vectorized
0.008213 seconds
```

In either R or Python, the vectorized version of random normal random variable (r.v.) is significantly faster than the scalar version. It is worth noting the usage of the `print(f'')` statement in the Python code, which is different from the way how we print the object of `Complex` class in chapter 1. In the code above, we use the `f−string`[28] which is a literal string prefixed with `'f'` containing expressions inside `{}` which would be replaced with their values. `f−string` was a feature introduced since Python 3.6. If you are familiar with Scala, you may find that this feature is quite similar with the string interpolation mechanism introduced since Scala 2.10.

It's also worth noting that lots of built-in functions in R are already vectorized, such as the basic arithmetic operators, comparison operators, `ifelse()`, element-wise logical operators `&,|`. But the logical operators `&&, ||` are not vectorized.

In addition to vectorization, there are also some built-in functions which may help to avoid the usages of for-loops. For example, in R we might be able use the `apply` family of functions to replace for-loops; and in Python the `map()` function can also be useful. In the Python `pandas` module, there are also many usages of `map/apply` methods. But in general the usage of `apply/map` functions has little or nothing to do with performance improvement. However, appropriate usages of such functions may help with the readability of the program. Compared with the `apply` family of functions in R, I think the `do.call()` function is more useful in practice. We would spend some time in `do.call()` later.

Considering the importance of vectorization in scientific programming, let's try to get more familiar with vectorization thorough the Biham–Middleton–Levine (BML) traffic model [29]. The BML model is very important in modern studies of traffic flow since it exhibits a sharp phase transition from free flowing status to a fully jammed status. A simplified BML model could be characterized as follows:

- Initialized on a 2-D lattice, each site of which is either empty or occupied by a colored particle (blue or red);
- Particles are distributed randomly through the initialization according to a uniform distribution; the two colors of particles are equally distributed.
- On even time steps, all blue particles attempt to move one site up and an attempt fails if the site to occupy is not empty;
- On Odd time steps, all red particles attempt to move one site right and an attempt fails if the site to occupy is not empty;
- The lattice is assumed periodic which means when a particle moves out of the lattice, it would move into the lattice from the opposite side.

The BML model specified above is implemented in both R/Python as follows to illustrate the usage of vectorization.

[28] https://www.python.org/dev/peps/pep-0498/
[29] https://en.wikipedia.org/wiki/Biham-Middleton-Levine_traffic_model

R

chapter2/BML.R

```
library(R6)
BML = R6Class(
  "BML",
  public = list(
    # alpha is the parameter of the uniform distribution to control particle
       distribution's density
    # m*n is the dimension of the lattice
    alpha = NULL,
    m = NULL,
    n = NULL,
    lattice = NULL,
    initialize = function(alpha, m, n) {
      self$alpha = alpha
      self$m = m
      self$n = n
      self$initialize_lattice()
    },
    initialize_lattice = function() {
      # 0 -> empty site
      # 1 -> blue particle
      # 2 -> red particle
      u = runif(self$m * self$n)
      # the usage of L is to make sure the elements in particles are of type
         integer;
      # otherwise they would be created as double
      particles = rep(0L, self$m * self$n)
      # doing inverse transform sampling
      particles[(u > self$alpha) &
                  (u <= (self$alpha + 1.0) / 2)] = 1L
      particles[u > (self$alpha + 1.0) / 2] = 2L
      self$lattice = array(particles, c(self$m, self$n))
    },
    odd_step = function() {
      blue.index = which(self$lattice == 1L, arr.ind = TRUE)
      # make a copy of the index
      blue.up.index = blue.index
      # blue particles move 1 site up
      blue.up.index[, 1] = blue.index[, 1] - 1L
      # periodic boundary condition
      blue.up.index[blue.up.index[, 1] == 0L, 1] = self$m
      # find which moves are feasible
      blue.movable = self$lattice[blue.up.index] == 0L
      # move blue particles one site up
      # drop=FALSE prevents the 2D array degenerates to 1D array
      self$lattice[blue.up.index[blue.movable, , drop = FALSE]] = 1L
      self$lattice[blue.index[blue.movable, , drop = FALSE]] = 0L
    },
    even_step = function() {
      red.index = which(self$lattice == 2L, arr.ind = TRUE)
      # make a copy of the index
      red.right.index = red.index
      # red particles move 1 site right
      red.right.index[, 2] = red.index[, 2] + 1L
      # periodic boundary condition
      red.right.index[red.right.index[, 2] == (self$n + 1L), 2] = 1
      # find which moves are feasible
      red.movable = self$lattice[red.right.index] == 0L
      # move red particles one site right
      self$lattice[red.right.index[red.movable, , drop = FALSE]] = 2L
      self$lattice[red.index[red.movable, , drop = FALSE]] = 0L
    }
  )
)
```

Now we can create a simple BML system on a 5×5 lattice using the R code above.

R

```
> source('BML.R')
> set.seed(2019)
> bml=BML$new(0.4,5,5)
> bml$lattice
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    0    2    1    1
[2,]    2    2    1    0    1
[3,]    0    0    0    2    2
[4,]    1    0    0    0    0
[5,]    0    1    1    1    0
> bml$odd_step()
> bml$lattice
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    0    2    1    0
[2,]    2    2    1    0    1
[3,]    1    0    0    2    2
[4,]    0    1    1    1    0
```

```
[5,]    0    0    0    0    1
> bml$even_step()
> bml$lattice
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    2    2    1    0
[2,]    2    2    1    0    1
[3,]    1    0    0    2    2
[4,]    0    1    1    1    0
[5,]    0    0    0    0    1
```

In the initialization step, we used the inverse transform sampling approach [30] to generate the status of each site. Inverse transform sampling method is basic but powerful approach to generate r.v. from any probability distribution given its cumulative distribution function (CDF). Reading the wiki page is enough to master this sampling method.

Python

```python
import numpy as np

class BML:
    def __init__(self, alpha, m, n):
        self.alpha = alpha
        self.shape = (m, n)
        self.initialize_lattice()

    def initialize_lattice(self):
        u = np.random.uniform(0.0, 1.0, self.shape)
        # instead of using default list, we use np.array to create the lattice
        self.lattice = np.zeros_like(u, dtype=int)
        # the parentheses below can't be ignored
        self.lattice[(u > self.alpha) & (u <= (1.0+self.alpha)/2.0)] = 1
        self.lattice[u > (self.alpha+1.0)/2.0] = 2

    def odd_step(self):
        # please note that np.where returns a tuple which is immutable
        blue_index = np.where(self.lattice == 1)
        blue_index_i = blue_index[0] - 1
        blue_index_i[blue_index_i < 0] = self.shape[0]-1
        blue_movable = self.lattice[(blue_index_i, blue_index[1])] == 0
        self.lattice[(blue_index_i[blue_movable],
                      blue_index[1][blue_movable])] = 1
        self.lattice[(blue_index[0][blue_movable],
                      blue_index[1][blue_movable])] = 0
```

[30] https://en.wikipedia.org/wiki/Inverse_transform_sampling

```

    def even_step(self):
        red_index = np.where(self.lattice == 2)
        red_index_j = red_index[1] + 1
        red_index_j[red_index_j == self.shape[1]] = 0
        red_movable = self.lattice[(red_index[0], red_index_j)] == 0
        self.lattice[(red_index[0][red_movable],
                      red_index_j[red_movable])] = 2
        self.lattice[(red_index[0][red_movable],
                      red_index[1][red_movable])] = 0
```

The Python implementation is also given.

R

```
>>> import numpy as np
>>> np.random.seed(2019)
>>> from BML import BML
>>> bml=BML(0.4,5,5)
>>> bml.lattice
array([[2, 0, 1, 1, 2],
       [0, 2, 2, 2, 1],
       [1, 0, 0, 2, 0],
       [2, 0, 1, 0, 2],
       [1, 1, 0, 2, 1]])
>>> bml.odd_step()
>>> bml.lattice
array([[2, 0, 0, 1, 2],
       [1, 2, 2, 2, 1],
       [0, 0, 1, 2, 0],
       [2, 1, 0, 0, 2],
       [1, 0, 1, 2, 1]])
>>> bml.even_step()
>>> bml.lattice
array([[0, 2, 0, 1, 2],
       [1, 2, 2, 2, 1],
       [0, 0, 1, 0, 2],
       [2, 1, 0, 0, 2],
       [1, 0, 1, 2, 1]])
```

Please note that although we have imported `numpy` in BML.py, we import it again in the code above in order to set the random seed. If we change the line to `from BML import *`, we don't need to import `numpy` again. But it is not recommended to `import *` from a module.

2.5 *Scope of Variables*

We have seen how to define variables in R/Python in chapter 1, and we have known that a variable is an identifier to a location in memory. What is the scope of a variable and why does it matter? Let's first have a look at the code snippets below.

R

```
> x=1
> var_func_1 = function(){print(x)
    }
> var_func_1()
[1] 1
> var_func_2 = function(){x=x+1;
    print(x)}
> var_func_2()
[1] 2
> x
[1] 1
```

Python

```
>>> x=1
>>> def var_func_1():print(x)
...
>>> var_func_1()
1
>>> def var_func_2():x+=1
...
>>> var_func_2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <
      module>
  File "<stdin>", line 1, in
      var_func_2
UnboundLocalError: local variable
    'x' referenced before
    assignment
```

The results of the code above seem strange before knowing the concept of variable scope. Inside a function, a variable may refer to a function argument/parameter or it could be formally declared inside the function which is called a local variable. But in the code above, `x` is neither a function argument nor a local variable. How does the `print()` function know where the identifier `x` points to?

The scope of a variable determines where the variable is available/accessible (can be referenced). Both R and Python apply lexical/static scoping for variables, which set the scope of a variable based on the structure of the program. In static scoping, when an 'unknown' variable referenced, the function will try to find it from the most closely enclosing block. That explains how the `print()` function could find the variable `x`.

In the R code above, `x=x+1` the first `x` is a local variable created by the = operator; the second `x` is referenced inside the function so the static scoping rule applies. As a result, a local variable `x` which is equal to 2 is created, which is independent with the `x` outside of the function `var_func_2()`. However, in Python when a variable is assigned a value in a statement the variable would treated as a local variable and that explains the `UnboundLocalError`.

Is it possible to change a variable inside a function which is declared outside the function? Based on the static scoping rule only, it's impossible. But there are workarounds in both R/Python. In R, we need the help of environment; and in Python we can use the keyword `global`.

So what is an environment in R? An environment is a place where objects are stored. When we invoke the interactive R session, an environment named as `.GlobalEnv` is created automatically. We can also use the function environment() to get the present environment. The ls() function can take an environment as the argument to list all objects inside the environment.

R

```
$r
> typeof(.GlobalEnv)
[1] "environment"
> environment()
<environment: R_GlobalEnv>
> x=1
> ls(environment())
[1] "x"
> env_func_1=function(x){
+   y=x+1
+   print(environment())
+   ls(environment())
+ }
> env_func_1(2)
<environment: 0x7fc59d165a20>
[1] "x" "y"
> env_func_2=function(){print(environment())}
> env_func_2()
<environment: 0x7fc59d16f520>
```

The above code shows that each function has its own environment containing all function arguments and local variables declared inside the function. In order to change a variable declared outside of a function, we need the access of the environment enclosing the variable to change. There is a function parent_env(e) that returns the parent environment of the given environment e in R. Using this function, we are able to change the value of x declared in .GlobalEnv inside a function which is also declared in .GlobalEnv. The global keyword in Python works in a totally different way, which is simple but less flexible.

R

```
> x=1
> env_func_3=function(){
+   cur_env=environment()
+   par_env=parent.env(cur_env)
+   par_env$x=2
+ }
> env_func_3()
> x
[1] 2
```

Python

```
>>> def env_func_3():
...   global x
...   x = 2
...
>>> x=1
>>> env_func_3()
>>> x
2
```

I seldomly use the `global` keyword in Python, if any. But the environment in R could be very handy in some occasions. In R, environment could be used as a purely mutable version of the `list` data structure.

R

```
# list is not purely mutable
> x=list(1)
> tracemem(x)
[1] "<0x7f829183f6f8>"
> x$a=2
> tracemem(x)
[1] "<0x7f828f4d05c8>"
```

R

```
# environment is purely mutable
> x=new.env()
> x
<environment: 0x7f8290aee7e8>
> x$a=2
> x
<environment: 0x7f8290aee7e8>
```

Actually, the object of an R6 class type is also an environment.

R

```
> # load the Complex class that we defined in chapter 1
> x = Complex$new(1,2)
> typeof(x)
[1] "environment"
```

In Python, we can assign values to multiple variables in one line.

Python

```
>>> x,y = 1,2
>>> x
1
>>> y
2
```

Python

```
>>> x,y=(1,2)
>>> print(x,y)
1 2
>>> (x,y)=(1,2)
>>> print(x,y)
1 2
>>> [x,y]=(1,2)
>>> print(x,y)
1 2
```

Even though in the left snippet above there aren't parentheses embracing `1,2` after the = operator, a tuple is created first and then the tuple is unpacked and assigned to `x, y`. Such mechanism doesn't exist in R, but we can define our own multiple assignment operator with the help of environment.

R

chapter2/multi_assignment.R

```r
`%=%` = function(left, right) {
  # we require the RHS to be a list strictly
  stopifnot(is.list(right))
  # dest_env is the desitination environment enclosing the variables on LHS
  dest_env = parent.env(environment())
  left = substitute(left)

  recursive_assign = function(left, right, dest_env) {
    if (length(left) == 1) {
      assign(x = deparse(left),
             value = right,
             envir = dest_env)
      return()
    }
    if (length(left) != length(right) + 1) {
      stop("LHS and RHS must have the same shapes")
    }

    for (i in 2:length(left)) {
      recursive_assign(left[[i]], right[[i - 1]],  dest_env)
    }
  }

  recursive_assign(left, right, dest_env)
}
```

Before going into the script deeper, first let's see the usage of the multiple assignment operator we defined.

R

```r
> source('multi_assignment.R')
> c(x,y,z) %=% list(1,"Hello World!",c(2,3))
> x
[1] 1
> y
[1] "Hello World!"
> z
[1] 2 3
> list(a,b) %=% list(1,as.Date('2019-01-01'))
> a
[1] 1
> b
[1] "2019-01-01"
```

In the `%=%` operator defined above, we used two functions `substitute, deparse` which are very powerful but less known by R novices. To better understand these functions as well as some other less known R functions, the Rchaeology [31] tutorial is worth reading.

It is also interesting to see that we defined the function `recursive_assign` inside the `%=%` function. Both R and Python support the concept of first class functions. More specifically, a function in R/Python is an object, which can be

1. stored as a variable;
2. passed as a function argument;
3. returned from a function.

The essential idea behind the `recursive_assign` function is a depth-first search (DFS), which is a fundamental graph traversing algorithm [32]. In the context of the `recursive_assign` function, we use DFS to traverse the parse tree of the `left` argument created by calling `substitute(left)`.

2.6 *Miscellaneous*

We have introduced the basics of R/Python programming so far. There are much more to learn to become an advanced user of R/Python. For example, the appropriate usages of `iterator, generator, decorator` could improve both the conciseness and readability of your Python code. The `generator` [33] is commonly seen in machine learning programs to prepare training/testing samples. `decorator` is a kind of syntactic sugar to allow the modification of a function's behavior in a simple way. In R there are no built-in `iterator , generator, decorator`, but you may find some third-party libraries to mimic these features; or you may try to implement your own.

One advantage of Python over R is that there are some built-in modules containing high-performance data structures or commonly-used algorithms implemented efficiently. For example, I enjoy using the `deque` structure in the Python `collections` module [34], but there is no built-in counterpart in R. We have written our own binary search algorithm earlier in this Chapter, which can also be replaced by the functions in the built-in module `bisect` [35] in Python.

Another important aspect of programming is testing. Unit testing is a typical software testing method that is commonly adopted in practice. In R there are two third-party packages `testthat` and `RUnit`. In Python, the built-in `unittest` is quite powerful. Also, the third-party module `pytest` [36] is very popular.

[31] https://cran.r-project.org/web/packages/rockchalk/vignettes/Rchaeology.pdf
[32] https://en.wikipedia.org/wiki/Depth-first_search
[33] https://docs.python.org/3/howto/functional.html
[34] https://docs.python.org/3/library/collections.html
[35] https://docs.python.org/3.7/library/bisect.html
[36] https://docs.pytest.org/en/latest/