# 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 Vector structure is actually immutable, compared with the mutable list structure in Python. A mutable object can be changed after it is created, but an immutable object can't.

You may have the question - If a Vector object can't be changed, why it is feasible to change the value of a Vector in R? To answer the question, first we have to understand the relation between variable and object in R/Python. A variable itself is a reference or pointer to an object (usually stored in the machine's memory). In R, when we try to modify a Vector, the original Vector is not modified at all. Instead, the variable to that Vector is linked to a new Vector object to reflect the modification. Let's verify it by checking the memory address.

*R*

```
> x=1:1000
> tracemem(x) # print the memory address of x whenever the internal code copies the object
[1] "<0x7f840927a240>"
> x[1]=-x[1]
tracemem[0x7f840927a240 -> 0x7f8408b2ae00]:
```

*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]=-x[0]
>>> hex(id(x))
'0x10592d908'
```

From the code snippet above, when we try to modify the value of `x[1]` in R, `x` points to a new object at address `0x7f8408b2ae00` (you probably would see different addresses on your machine). But in Python, the memory address doesn't change after we change the value of the first element because list in Python is mutable. 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 [4].

### 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
```

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

```
[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
>>> 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 [5]. 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
```

list in R is also immutable, like Vector. 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.

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

*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 (also because list in R is immutable). 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 [6] is an R package and pandas [7] 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

[6] https://cran.r-project.org/web/packages/data.table/index.html
[7] https://pandas.pydata.org/

[8] 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 [9]. We can also do functional programming (FP) and OOP in R/Python. R is more like a functional programming language than Python as most of the built-in data structures in R are immutable. 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 [10] 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$real=real
+ },
+ # define a function to change the imag value
+ set_imag = function(imag){
+   self$imag=imag
+ },
+ # override print function
+ print = function(){
```

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

```
+    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 [11].

You may be curious of the double underscore surrounding the methods, such as `__init__` and `__repr__`. These methods are well-known as magic methods [12]. 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.

*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)
> 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 [13].

[11] https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
[12] https://rszalski.github.io/magicmethods/
[13] https://docs.python.org/3.3/reference/lexical_analysis.html

```
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 [14]. 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` [15] to install modules in Python

### 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.

[14] https://cran.r-project.org/doc/manuals/r-release/R-lang.html#Identifiers

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

### <- vs. =

If you have known R before, you probably heard of the advice [16] 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
```

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

# 2

# *More on R/Python Programming*