2.7 Code Examples: R & Python

Below we present some examples of equivalent code in R and Python for easier comparison. We note that the subsections dedicated to R and Python should be studied beforehand to get the general overview of the programming languages as this chapter basically summarizes the fuctionality and provides a side-by-side comparison for select operations. Again, this is not an exhaustive comparison and any additional operations, which may be needed, will be covered in their relevant topics.

You can get additional information on many functions in either `R` of `Python`:

- **in R**:

You can use `?function_name` or `help(function_name)` to get the help about any function, for example:

```
?print
```

```
## Print Values
##
## Description:
##
##      'print' prints its argument a
##      'invisible(x)').  It is a gen
##      printing methods can be easil
##
## Usage:
##
##      print(x, ...)
##
## ...
```

- **in Python**:

You can use `help(function_name)` to get help about any function, for example:

```
help(print)
```

```
## Help on built-in function print in
##
## print(...)
##     print(value, ..., sep=' ', end
##
##     Prints the values to a stream,
##     Optional keyword arguments:
##     file:  a file-like object (str
##     sep:   string inserted between
##     end:   string appended after t
##     flush: whether to forcibly flu
```

Note that the documentation quality will vary (e.g. some functions may have a more detailed explanations and examples in `R` compared to similar ones in `Python` and vice versa).

2.7.1 Working with data arrays

We begin by examining a data array.

We can create a simple vector with some values:

- **in R**:

```r
#Create a vector:
my_vec <- c(1, 2, 3, 4, 5, 6, 7, 8)
str(my_vec)
```

```
##  num [1:8] 1 2 3 4 5 6 7 8
```

- **in Python**:

```python
#Create a list:
my_vec = [1, 2, 3, 4, 5, 6, 7, 8]
print(type(my_vec))
```

```
## <class 'list'>
```

We can select specific elements. Note that in `Python`, the values start at index `0`, while in `R` they start at index `1`:

```r
#Select the data:
print(my_vec[1])
```

```
## [1] 1
```

```r
print(my_vec[8])
```

```
## [1] 8
```

```r
print(my_vec[1:3])
```

```
## [1] 1 2 3
```

```r
print(my_vec[c(1, 2, 4, 8)])
```

```
## [1] 1 2 4 8
```

```python
#Select the data:
print(my_vec[0])
```

```
## 1
```

```python
print(my_vec[7])
```

```
## 8
```

```python
print(my_vec[0:3])
```

```
## [1, 2, 3]
```

```python
print([my_vec[i] for i in [0, 1, 3, 7
```

```
## [1, 2, 4, 8]
```

Note that when selecting multiple values, in `R` the index range is `1:3 = {1, 2, 3}`, while in `Python` the index range `0:3 = {0, 1, 2}` - i.e. the last value is not included. Similarly, we can also create a range of values:

```r
my_range <- seq(from = 1, to = 4, by
print(my_range)
```

```
## [1] 1 2 3 4
```

```r
print(my_range[1:4])
```

```
## [1] 1 2 3 4
```

```python
my_range = list(range(1, 5))
print(my_range)
```

```
## [1, 2, 3, 4]
```

```python
print(my_range[1:4])
```

```
## [2, 3, 4]
```

We can get the length of the data array:

```r
print(length(my_vec))
```

```
## [1] 8
```

```python
print(len(my_vec))
```

```
## 8
```

We can loop through our data array and print the elements:

```r
for(i in 1:length(my_vec)){
  print(my_vec[i])
}
```

```python
for i in range(0, len(my_vec)):
  print(my_vec[i])
#
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
```

```
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
```

We can select every **3nd** element, starting from the **2nd** element:

```r
print(my_vec[seq(2, length(my_vec), 3
```

```python
print(my_vec[1::3])
```

```
## [1] 2 5 8
```

```
## [2, 5, 8]
```

Or every **2nd** element starting from the **1st**:

```r
print(my_vec[seq(1, length(my_vec), 2
```

```python
print(my_vec[0::2])
```

```
## [1] 1 3 5 7
```

```
## [1, 3, 5, 7]
```

We can **add** two vectors together:

```r
my_vec_1 = seq(from = 1, to = 5, by =
my_vec_2 = rev(my_vec_1)
cat("vec. 1: ", my_vec_1)
```

```
## vec. 1:  1 2 3 4 5
```

```r
cat("vec. 2: ", my_vec_2)
```

```
## vec. 2:  5 4 3 2 1
```

```r
vec_sum = my_vec_1 + my_vec_2
cat("vec. 1 + 2: ", vec_sum)
```

```
## vec. 1 + 2:  6 6 6 6 6
```

```python
my_vec_1 = list(range(1, 6))
my_vec_2 = my_vec_1[::-1]
print("vec. 1: ", my_vec_1)
```

```
## vec. 1:  [1, 2, 3, 4, 5]
```

```python
print("vec. 2: ", my_vec_2)
```

```
## vec. 2:  [5, 4, 3, 2, 1]
```

```python
vec_sum = [a + b for a, b in zip(my_v
print("vec. 1 + 2: ", vec_sum)
```

```
## vec. 1 + 2:  [6, 6, 6, 6, 6]
```

In this case, we use `zip` to create pairs from the two lists:

```python
print(list(zip(my_vec_1, my_vec_2)))
```

```
## [(1, 5), (2, 4), (3, 3), (4, 2), (5, 1)]
```

Then we iterate through the pairs and sum the elements in each pair.

Note that the addition of two vectors `my_vec_1 + my_vec_2` in R has a different meaning in Python:

```r
cat("vec. 1 & 2: ", c(my_vec_1, my_ve
```

```r
## vec. 1 & 2:  1 2 3 4 5 5 4 3 2 1
```

```python
print("vec. 1 & 2: ", my_vec_1 + my_v
```

```python
## vec. 1 & 2:  [1, 2, 3, 4, 5, 5, 4,
```

If we use `numpy.array` instead of a list, then we can add the two vectors together in Python like we do in R:

```python
import numpy as np

my_vec_1_np = np.array(my_vec_1)
np_vec_2_np = np.array(my_vec_2)
print("vec. 1 + 2: ", my_vec_1_np + np_vec_2_np)
```

```
## vec. 1 + 2:  [6 6 6 6 6]
```

We can multiply the vector elements by a constant:

```r
cat(my_vec_1 * 3)
```

```r
## 3 6 9 12 15
```

```python
print([x * 3 for x in my_vec_1])
```

```python
## [3, 6, 9, 12, 15]
```

Again, for a `list` in Python, the `*` has a different meaning:

```r
cat(rep(my_vec_1, 3))
```

```
## 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```r
print(my_vec_1 * 3)
```

```
## [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1,
```

Though if we use `numpy.array`, then it works in a similar way as in `R`:

```python
import numpy as np

my_vec_1_np = np.array(my_vec_1)
print(my_vec_1_np * 3)
```

```
## [ 3  6  9 12 15]
```

2.7.2 Working with strings

Similarly, we can create character strings. `R` has specific functions to modify strings, while `Python` has these modules implemented in the string class itself, so we always know which functions are available by using the dot `.` operator and pressing `Tab` in the editor to get the full list of modules for a specific class:

- in R:

```r
my_animal = "bear"
print(my_animal)
```

```
## [1] "bear"
```

```r
print(toupper(my_animal))
```

```
## [1] "BEAR"
```

```r
my_sentence = paste("I", "saw", "a",
print(my_sentence)
```

```
## [1] "I saw a bear"
```

```r
print(paste("----", "++++", sep = "")
```

```
## [1] "----++++"
```

- in Python:

```python
my_animal = "bear"
print(my_animal)
```

```
## bear
```

```python
print(my_animal.upper())
```

```
## BEAR
```

```python
my_sentence = "I " + "saw a " + my_an
print(my_sentence)
```

```
## I saw a bear
```

```python
print("----" + "++++")
```

```
## ----++++
```

2.7.3 Variables containing different data type values

It is often desireable to have a variable contain different types of information - integer, string, boolean values.

- in R:

```r
my_data <- list(name = "Joe",
                grades = c(8, 7, 9),
                has_attended = TRUE)
str(my_data)
```

```
## List of 3
##  $ name        : chr "Joe"
##  $ grades      : num [1:3] 8 7 9
##  $ has_attended: logi TRUE
```

- in Python:

```python
my_data = {"name": "Joe",
           "grades": [8, 7, 9],
           "has_attended": True}
print(type(my_data))
```

```
## <class 'dict'>
```

```python
print(my_data)
```

```
## {'name': 'Joe', 'grades': [8, 7, 9
```

```r
str(my_data["name"])
```

```
## List of 1
##  $ name: chr "Joe"
```

```r
print(my_data["name"])
```

```
## $name
## [1] "Joe"
```

```r
str(my_data[["name"]])
```

```
##  chr "Joe"
```

```r
print(my_data[["name"]])
```

```
## [1] "Joe"
```

```r
print(type(my_data["name"]))
```

```
## <class 'str'>
```

```r
print(my_data["name"])
```

```
## Joe
```

```r
str(my_data["grades"])
```

```
## List of 1
##  $ grades: num [1:3] 8 7 9
```

```r
str(my_data["has_attended"])
```

```
## List of 1
##  $ has_attended: logi TRUE
```

```r
print(my_data[["grades"]])
```

```
## [1] 8 7 9
```

```r
print(my_data$has_attended)
```

```
## [1] TRUE
```

```python
print(type(my_data["grades"]))
```

```
## <class 'list'>
```

```python
print(type(my_data["has_attended"]))
```

```
## <class 'bool'>
```

```python
print(my_data["grades"])
```

```
## [8, 7, 9]
```

```python
print(my_data["has_attended"])
```

```
## True
```

As we can see, these variales are able to house value of different types. In `R` we can select the relevant values in a number of different ways.

Often, however, we have more than one observation with different properties (e.g. a database of people with names, unique ID's, email addreses, indicator value if it is a new member, etc.) and we want to have a matrix-like structure (i.e. a table) to house those values.

```r
my_dataset = data.frame(name = c("Joh
                        wage = c(800,
                        is_employed =
                        stringsAsFact
str(my_dataset)
```

```
## 'data.frame':    3 obs. of  3 vari
##  $ name       : chr  "John" "Sam"
##  $ wage       : num  800 600 700
##  $ is_employed: logi  TRUE TRUE FA
```

```r
print(my_dataset)
```

```
##   name wage is_employed
## 1 John  800        TRUE
## 2  Sam  600        TRUE
## 3  Tim  700       FALSE
```

Note, the `stringsAsFactors` is required so that the `name` column would be a character vector instead of a factor (a factor is a vector of **integer values** with a corresponding set of **character values** to use when the factor is **displayed**).

```python
import pandas as pd
#
my_dataset = {"name": ["John", "Sam",
              "wage": [800, 600, 700]
              "is_employed": [True, T
#
my_dataset = pd.DataFrame(my_dataset,
print(type(my_dataset))
```

```
## <class 'pandas.core.frame.DataFram
```

```python
print(my_dataset)
```

```
##    name  wage  is_employed
## 0  John   800         True
## 1   Sam   600         True
## 2   Tim   700        False
```

```python
print(my_dataset.keys())
```

```
## Index(['name', 'wage', 'is_employe
```

The `columns = my_dataset.keys()` is required to preserve the order of the columns.

P.S. The term `panel data` is derived from econometrics and is partially responsible for the name `pandas`: `pan(el)-da(ta)-s`.

Values can be accessed directly:

```r
print(my_dataset["name"])
```

```
##   name
## 1 John
## 2  Sam
## 3  Tim
```

```python
print(my_dataset["name"])
```

```
## 0    John
## 1     Sam
## 2     Tim
## Name: name, dtype: object
```

```r
print(my_dataset$is_employed)
```

```
## [1]  TRUE  TRUE FALSE
```

```python
print(my_dataset["is_employed"])
```

```
## 0     True
## 1     True
## 2    False
## Name: is_employed, dtype: bool
```

Note that using `$` in `R` returns a vector, while specifying `["name"]` returns a `data.frame` with one columns (as evident by the output format).

2.7.4 Defining Functions

We can also define our own functions.

Let's define a simple function, which compares two values:

- **in R:**
- **in Python:**

```r
#Define a function:
my_compare <- function(x, y){
  if(x < y){
    print("1st value is smaller")
  }else{
    if(x > y){
      print("1st value is greater")
    }else{
      print("Values are equal")
    }
  }
}
#Test the function
my_compare(1, 1)
```

```
## [1] "Values are equal"
```

```r
my_compare(1, 2)
```

```
## [1] "1st value is smaller"
```

```r
my_compare(2, 1)
```

```
## [1] "1st value is greater"
```

```python
#Define a function:
def my_compare(x, y):
  if x < y:
    print("1st value is smaller")
  elif x > y:
    print("1st value is greater")
  else:
    print("Values are equal")
#
#
#
#
#Test the function
my_compare(1, 1)
```

```
## Values are equal
```

```python
my_compare(1, 2)
```

```
## 1st value is smaller
```

```python
my_compare(2, 1)
```

```
## 1st value is greater
```

Let's say we want to define a custom addition function, which increases the values of two elements by one before adding them together:

```r
#Define a function:
my_sum <- function(x, y){
  x <- x + 1
  y <- y + 1
  return(x + y)
}
#Test the function
print(my_sum(1, 2))
```

```
## [1] 5
```

```python
#Define a function:
def my_sum(x, y):
  x = x + 1
  y = y + 1
  return x + y
#
#Test the function
print(my_sum(1, 2))
```

```
## 5
```

We can also create a function which creates a summary of a data array:

```r
#Define a function:
my_summary <- function(x){
  min_val = min(x)
  max_val = max(x)
  sum_val = sum(x)
  avg_val = mean(x)
  output = list(min = min_val,
                max = max_val,
                average = avg_val,
                sum = sum_val)
  return(output)
}
#Test the function
my_results = my_summary(c(-2, 1, 5))
print(my_results$min)
```

```
## [1] -2
```

```r
print(my_results[["average"]])
```

```python
import numpy as np
#Define a function:
def my_summary(x):
  min_val = min(x)
  max_val = max(x)
  sum_val = sum(x)
  avg_val = np.mean(x)
  output = {"min": min_val,
            "max": max_val,
            "average": avg_val,
            "sum": sum_val}
  return output
#Test the function
my_results = my_summary([-2, 1, 5])
print(my_results["min"])
```

```
## -2
```

```python
print(my_results["average"])
```

```
## [1] 1.333333
```

```
## 1.3333333333333333
```

```
print(str(my_results))
```

```
print(type(my_results))
```

```
## List of 4
##  $ min    : num -2
##  $ max    : num 5
##  $ average: num 1.33
##  $ sum    : num 4
## NULL
```

```
## <class 'dict'>
```

We note the different data types - a list in `R` and a dictionary in `Python`. We can also use the `print` function with the whole variable:

```
print(my_results)
```

```
print(my_results)
```

```
## $min
## [1] -2
##
## $max
## [1] 5
##
## $average
## [1] 1.333333
##
## $sum
## [1] 4
```

```
## {'min': -2, 'max': 5, 'average': 1
```

2.7.5 Lists in Python are *mutable*

- **in R**:

```r
my_list <- function(a_vec){
  a_vec[1] <- "88"
  append(a_vec, "!!!")
}
#Test the function
old_vec <- c("a", "b")
cat("Old: ", old_vec, "\n")
```

```
## Old:  a b
```

```r
new_vec <- my_list(old_vec)
cat("New: ", new_vec, "\n")
```

```
## New:  88 b !!!
```

```r
cat("Old: ", old_vec, "\n")
```

```
## Old:  a b
```

- **in Python**:

```python
def my_list(a_list):
  a_list[0] = "88"
  a_list.append("!!!")
  return a_list
#Test the function
old_list = ["a", "b"]
print("Old: " + str(old_list))
```

```
## Old: ['a', 'b']
```

```python
new_list = my_list(old_list)
print("New: " + str(new_list))
```

```
## New: ['88', 'b', '!!!']
```

```python
print("Old: " + str(old_list))
```

```
## Old: ['88', 'b', '!!!']
```

The original values **were changed in Python**, even though we assigned the function output to a new variable!

In order to **not modify** the object we are passing, we can create a new *reference* inside our function:

```
def my_list(a_list):
  #Create a new reference:
  #Method 1
  #b_list = a_list[:]
  #Method 2
  b_list = list(a_list)
  #Modify the values:
  b_list[0] = "88"
  b_list.append("!!!")
  return b_list
#Test the function
old_list = ["a", "b"]
print("Old: " + str(old_list))
```

```
## Old: ['a', 'b']
```

```
new_list = my_list(old_list)
print("Old: " + str(old_list))
```

```
## Old: ['a', 'b']
```

```
print("New: " + str(new_list))
```

```
## New: ['88', 'b', '!!!']
```

Note that if we write `b_list = a_list` instead of `b_list = a_list[:]` (or instead of `b_list = list(a_list)`), then we will again **modify the original variable**!

Newertheless, it is still useful to pass **by reference** instead of **by value** if we do not want our function to return anything but still change the original values.

2.7.6 Creating Matrices

We will create the following matrix:

$$\mathbf{X} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

```r
x1 <- c(1, 2, 3)
x2 <- c(4, 5, 6)
x  <- cbind(x1, x2)
print(x) # the vectors are combined a
```

```
##      x1 x2
## [1,]  1  4
## [2,]  2  5
## [3,]  3  6
```

```python
import numpy as np
#
x1 = [1, 2, 3]
x2 = [4, 5, 6]
x_cs = np.column_stack((x1, x2))
x_vs = np.vstack((x1, x2))
print(x_cs)  # a 2-dimensional array
```

```
## [[1 4]
##  [2 5]
##  [3 6]]
```

```python
print(x_vs)  # a 2-dimensional array
```

```
## [[1 2 3]
##  [4 5 6]]
```

Note that we can transpose the matrix (so the column elements become the row elements and vice versa) in the following way - using `t(...)` in `R` and `np.transpose(...)` in `Python`:

```r
print(t(x))
```

```
##    [,1] [,2] [,3]
## x1    1    2    3
## x2    4    5    6
```

```python
print("Column-stacked lists:\n", x_cs
```

```
## Column-stacked lists:
##  [[1 4]
##  [2 5]
##  [3 6]]
```

```
print("Transposed row-stacked lists:\
```

```
## Transposed row-stacked lists:
##  [[1 4]
##  [2 5]
##  [3 6]]
```

```
print("Row-stacked lists:\n", x_vs)
```

```
## Row-stacked lists:
##  [[1 2 3]
##  [4 5 6]]
```

```
print("Transposed column-stacked list
```

```
## Transposed column-stacked lists:
##  [[1 2 3]
##  [4 5 6]]
```

Note the different ways that the matrix is constructed in `Python` - using `np.column_stack` creates $\mathbf{X}$, whereas using `np.vstack` creates $\mathbf{X}^\top$.

We can access different elements from the matrix $\mathbf{X}$:

```
print(x[1, ]) # print first row, to r
```

```
print("Column-stacked lists:")
```

```
## x1 x2
##  1  4
```

```
print(x[, 1]) # print first column
```

```
## [1] 1 2 3
```

```
print(x[1, 1]) # x_{1,1}
```

```
## x1
##  1
```

```
print(x[3, 2]) # x_{3,2}
```

```
## x2
##  6
```

```
## Column-stacked lists:
```

```
print(x_cs[0])                  # print
```

```
## [1 4]
```

```
print(np.transpose(x_cs)[0])  # print
```

```
## [1 2 3]
```

```
print(x_cs[0][0]) # X_{1,1}
```

```
## 1
```

```
print(x_cs[2][1]) # X_{3,2}
```

```
## 6
```

```
print("Row-stacked lists:")
```

```
## Row-stacked lists:
```

```
print(np.transpose(x_vs)[0])  # print
```

```
## [1 4]
```

```
print(x_vs[0])                # print
```

```
## [1 2 3]
```

```
print(x_vs[0][0]) # X_{1,1}
```

```
## 1
```

```
print(x_vs[1][2]) # X_{3,2}
```

```
## 6
```

We can multiply different matrices:

$$\mathbf{X}^\top\mathbf{X} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 14 & 32 \\ 32 & 77 \end{bmatrix}$$

```
print(t(x) %*% x)
```

```
##    x1 x2
## x1 14 32
## x2 32 77
```

```
print(np.dot(np.transpose(x_cs), x_cs
```

```
## [[14 32]
##  [32 77]]
```

```
print(np.dot(x_vs, np.transpose(x_vs)
```

```
## [[14 32]
##  [32 77]]
```

Note that using `np.column_stack` allows us to implement the formula as it is written, i.e. by transposing the first matrix.

2.7.7 Classes in R and Python

Note: there are so called `S3`, `S4` and `Reference` classes in `R`, though their use depends on the individual package creator. For some tutorials on creating classes in `R`, see this page and a mini-examples for both S4 and Reference Classes.

We will restrict ourselves to a simple example for both `R` and `Python`:

- Our class will have 3 variables: `x`, `y` and `z`;
- Our class will have a function to print `hello`;
- Our class will have a function to double the value(s) of `y`;
- Our class will have a function which increases the value of the passed variable by 1;

- **in R**:

To define a class, you call `setRefClass`. The first argument is the name of the class, and by convention this should be the same as the variable that you assign the result to. You also need to pass lists to the arguments “fields” and “methods”.

- **in Python**:

The `__init__` is a method that sets the values for any parameters that need to be defined when an object of this class is first created. The `self.` part is a syntax that allows access to a variable from anywhere else in the class:

```r
MyClass <- setRefClass(
  "MyClass",
  fields = list(
    x = "ANY", #any kind of variable
    y = "numeric", #integer variable
    z = "character"#char variable typ
  ),
  methods = list(
    initialize = function(x = NULL, y
      #Note the default values for x,
      #This method is called when you
      x <<- x
      y <<- y
      z <<- z
      print("You initialized MyClass!
    },
    hello = function(){
      #This method returns the string
      return("hello")
    },
    doubleY = function(){
      return(2 * y)
    },
    mySum = function(input){
      return(input + 1)
    }
  )
)

#Create a new instance of the class:
my_obj = MyClass$new(x = NULL, y = 1:
```

```python
import numpy as np
import string
#
#
class MyClass():
  #
  def __init__(self, x = None, y = No
    #This method is called when you c
    #Specify the default values:
    if y is None:
      y = list(range(1, 11))
    if z is None:
      z = string.ascii_lowercase #alp
    self.y = y
    self.x = x
    self.z = z
    print("You initialized MyClass!")
  #
  #Since class methods are not functi
  def hello(self):
    return "hello"
  #
  def doubleY(self):
    return 2 * np.array(self.y)
  #
  def mySum(self, my_input):
    return np.array(my_input) + 1
#
#
#Create a new instance of the class:
my_obj = MyClass(x = None, y = list(r
```

```
## [1] "You initialized MyClass!"
```

```
## You initialized MyClass!
```

```r
print(my_obj)
```

```
## Reference class object of class "M
## Field "x":
## NULL
## Field "y":
##  [1]  1  2  3  4  5  6  7  8  9 10
## Field "z":
## [1] "a"  "b"  "cd"
```

```python
print(my_obj)
```

```
## <__main__.MyClass object at 0x0000
```

Now we can access the values and functions in the initialized class objects:

```r
print(my_obj$x)
```

```
## NULL
```

```python
print(my_obj.x)
```

```
## None
```

```r
print(my_obj$y)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```python
print(my_obj.y)
```

```
## [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```r
print(my_obj$z)
```

```
## [1] "a"  "b"  "cd"
```

```python
print(my_obj.z)
```

```
## ['a', 'b', 'cd']
```

```r
my_obj$hello()
```

```
## [1] "hello"
```

```python
print(my_obj.hello())
```

```
## hello
```

```r
my_obj$doubleY()
```

```
##  [1]  2  4  6  8 10 12 14 16 18 20
```

```python
print(my_obj.doubleY())
```

```
## [ 2  4  6  8 10 12 14 16 18 20]
```

```r
my_obj$mySum(5)
```

```
## [1] 6
```

```python
print(my_obj.mySum(5))
```

```
## 6
```

2.7.8 Plotting data

Plotting data is very similar in both `R` and `Python`:

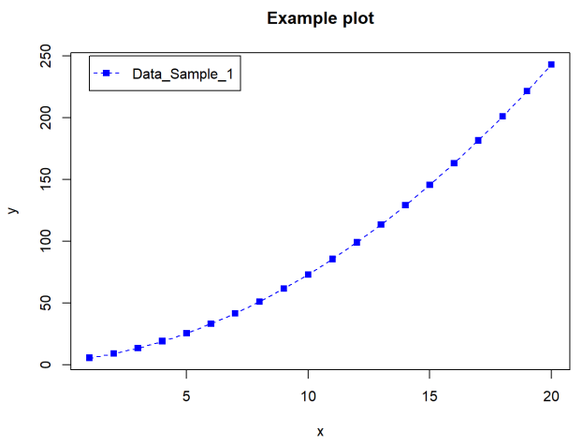
- in R:

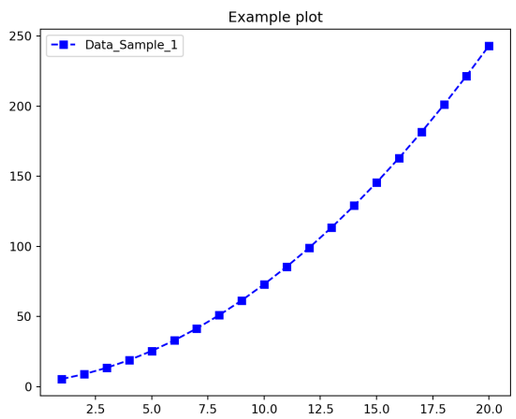
```
#


x <- 1:20
y <- 0.5 * x^2 + 2 * x + 3
#
plot(x, y,
     type = "l", lty = 2, col = "blue
points(x, y, col = "blue", pch = 15)
title(main = "Example plot")
legend(x = 1, y = 250,
       legend = c("Data_Sample_1"),
       lty = 2, pch = 15, col = "blue
```

- in Python:

```
import matplotlib.pyplot as plt
#
x = list(range(1, 21))
y = [0.5 * a**2 + 2 * a + 3 for a in
#
plt.figure(0)
plt.plot(x, y,
        linestyle = "--", color = "bl
        marker = "s", #squares
        label = "Data_Sample_1")
plt.title("Example plot")
plt.legend(loc = "upper left")
plt.show()
```



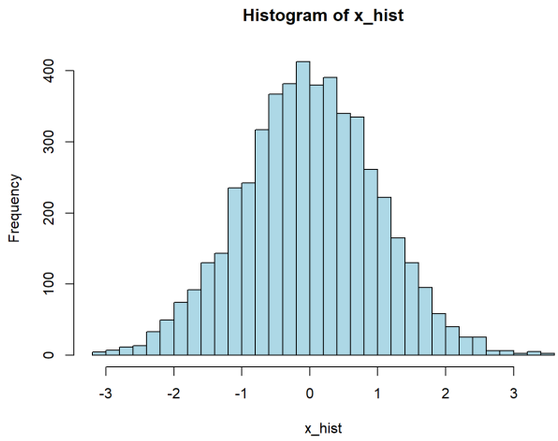


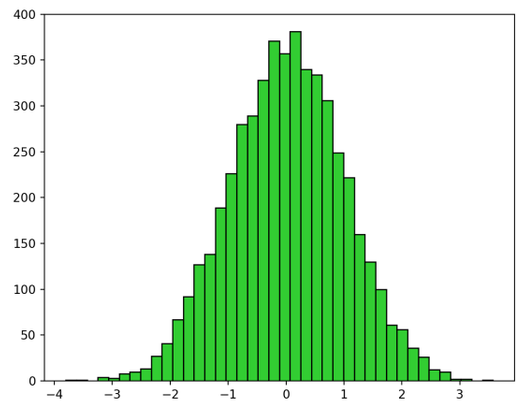
```r
#

set.seed(123)
nsample <- 5000
x_hist <- rnorm(nsample)

hist(x_hist, breaks = 40, col = "ligh
```

```python
import numpy as np
import matplotlib.pyplot as plt
#
np.random.seed(123)
nsample = 5000
x_hist = np.random.normal(size = nsam
plt.figure(1)
tmp_hist_ret = plt.hist(x_hist, bins
                        color = "limegreen"
plt.show()
```



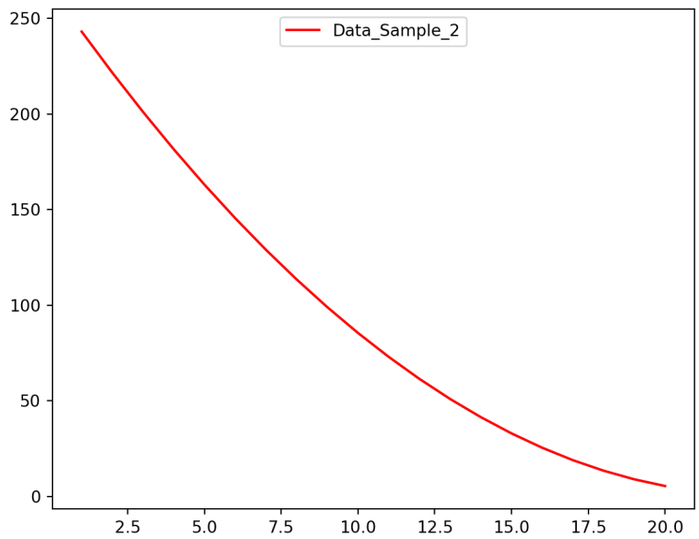


Using `plt.figure()` we can number the plots in `Python` in order to separate them as well as edit them if we ever need to by specifying `plt.figure(number)`, where the `number` is the ID of the figure that we want to edit (or create a new figure).

Note that `matplotlib.pyplot.hist()` returns a tuple with the value of histogram bins, `n`, the edges of the `bins` and a list of individual `patches` used to create the histogram. In some IDE's, these returned values are not printed by default but for the notes in this book, we need to supress them manually.

If we need to access the plot figure that we created previously then, as long as we did not delete it from the working environment, we can select it using the `plt.figure` method:

```
y2 = y[::-1]
_ = plt.figure(0)
_ = plt.plot(x, y2, linestyle = "-", color = "red", label = "Data_Sample_2")
_ = plt.legend(loc = "upper center")
plt.show()
```



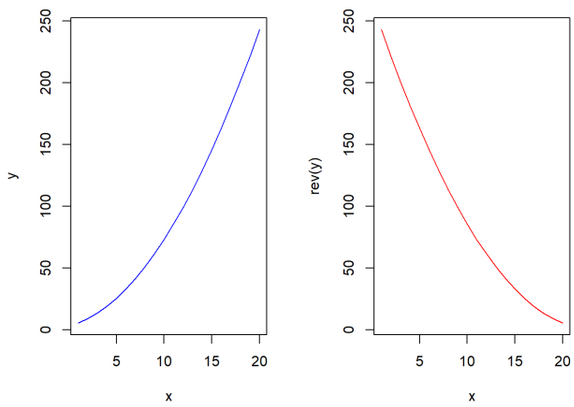
We can also plot multiple figures:

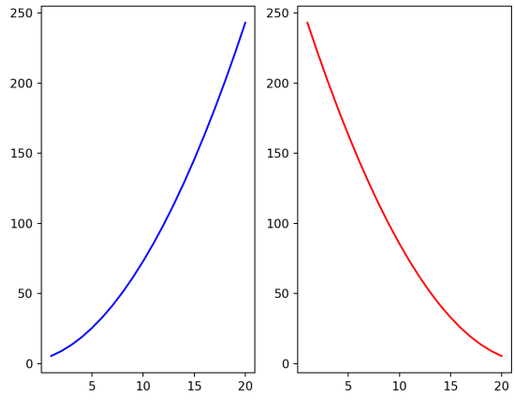
We can specify a 1-row, 2-column layout using `par(mfrow = c(1, 2))` :

We can specify the 1-row, 2-column layout using `add_subplot(1, 2, c)` or `add_subplot(12c)` where `c` is the position (integer number) of the plot (`c = 1` - plot in the first layout space, `c = 2` - plot in the second layout space).

```r
#
#
par(mfrow = c(1, 2))
plot(x, y, type = "l", col = "blue")
plot(x, rev(y), type = "l", col = "re
```

```python
_ = plt.figure(2).add_subplot(121)
_ = plt.plot(x, y, color = "blue")
_ = plt.figure(2).add_subplot(122)
_ = plt.plot(x, y2, color = "red")
plt.show()
```





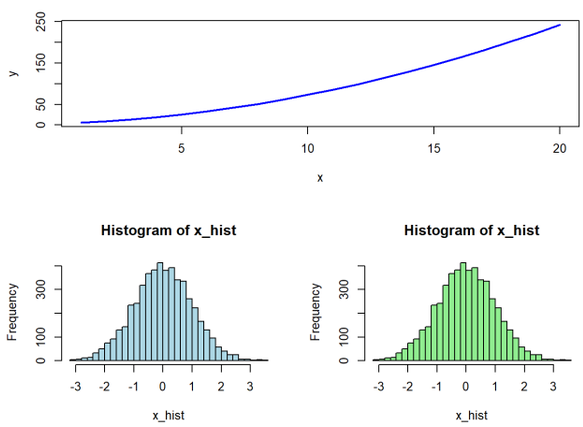
We can also plot an odd number of plots:

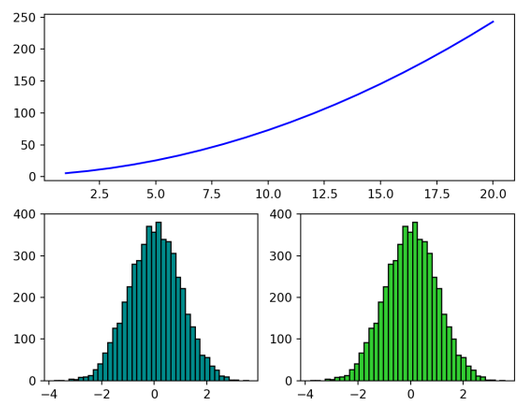
By using the `layout` function, we can specify a matrix layout of how we want our plots positioned, where the number indicates the plot number - the larger the matrix, the more precide our positioning can be. A `0` indicates to not plot at that position.

Using `add_subplot(abc)` we can specify either an odd or even number of plots by specifying a different subplot layouts (e.g. the same number of rows but different columns) for some of the plots, but their positions must not overlap!

```r
my_matrix = matrix(c(1, 1,
                     2, 3), nrow = 2,
#
#
#
layout(my_matrix)
plot(x, y, type = "l", col = "blue",
hist(x_hist, breaks = 40, col = "ligh
hist(x_hist, breaks = 40, col = "ligh
```

```python
_ = plt.figure(3).add_subplot(211)
_ = plt.plot(x, y, color = "blue")
_ = plt.figure(3).add_subplot(223)
tmp_hist_ret = plt.hist(x_hist, color
                        bins = 40, edgecolo
_ = plt.figure(3).add_subplot(224)
tmp_hist_ret = plt.hist(x_hist, color
                        bins = 40, edgecolo
plt.show()
```





2.7.9 Advanced plot libraries

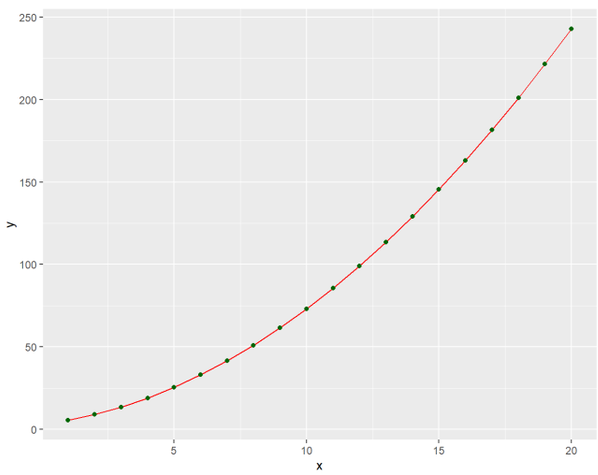
There are also advanced plot capabilities in both R (e.g. `ggplot2`, `plotly`) and Python (e.g. `ggplot2`, `Bokeh`, `plotly`). Some examples for `ggplot2` can be found here.

In `R` use `install.packages("ggplot2")` to install the package. For `Python` the package is called `plotnine` - in Anaconda Navigator use `conda install -c conda-forge plotnine`:

```r
library(ggplot2)
```

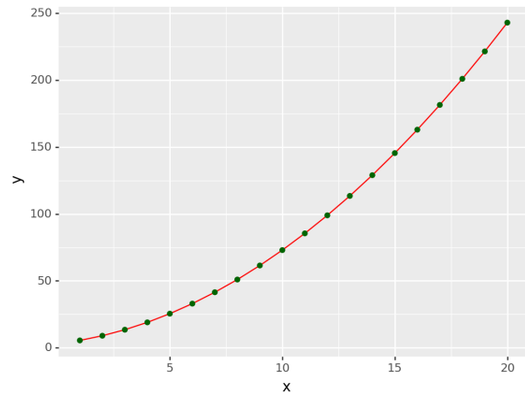
```
## Warning: package 'ggplot2' was bui
```

```r
data <- data.frame(x, y)
p <- ggplot(data, aes(x = x, y = y))
p <- p + geom_line(col = "red") + geo
p
```




```python
from plotnine import *
#
data = pd.DataFrame(data = {"x": x, "
p = ggplot(data, aes(x = 'x', y = 'y'
p = p + geom_line(color = "red") + ge
print(p)
```

```
## <ggplot: (133773328)>
```




```python
del p # Remove the variable from Pyth
```