Python & R: a Short Overview and Comparison

Andrius Buteikis, andrius.buteikis@mif.vu.lt
http://web.vu.lt/mif/a.buteikis/

Installing and Using Python

A very brief introduction

A general comparison of R and Python and

an article giving an overview.

Installing R and Python

See the lecture notes (Chapter "2 Software") on how to install `R`, `Python` and their main packages.

Introductory Tutorials

- For `R`: use the `swirl` package to install `swirl::install_course("R Programming")` and run the tutorial as per the lecture notes;
- For `Python` download `PyCharm Edu` as per the lecture notes.

Additional Python tutorials (optional)

1. Basic tutorial
2. Some Statistical model estimation examples
3. More advanced topics

Main IDE's:

When you launch the Anaconda Navigator, the following screen appears:

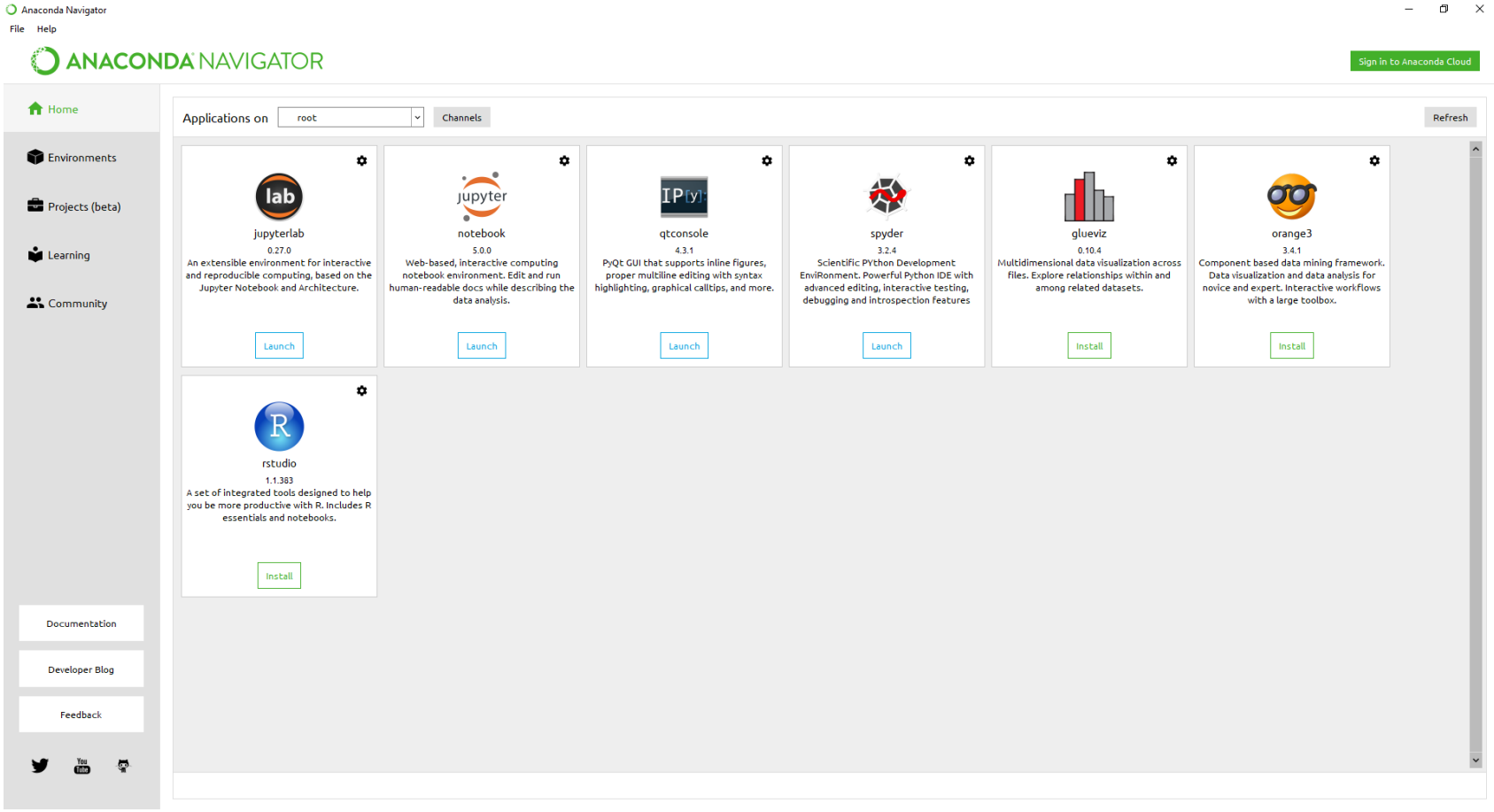


Figure 1: Anaconda Navigator window

Spyder

Visually very similar to RStudio. The development of Spyder might slow down in the future.

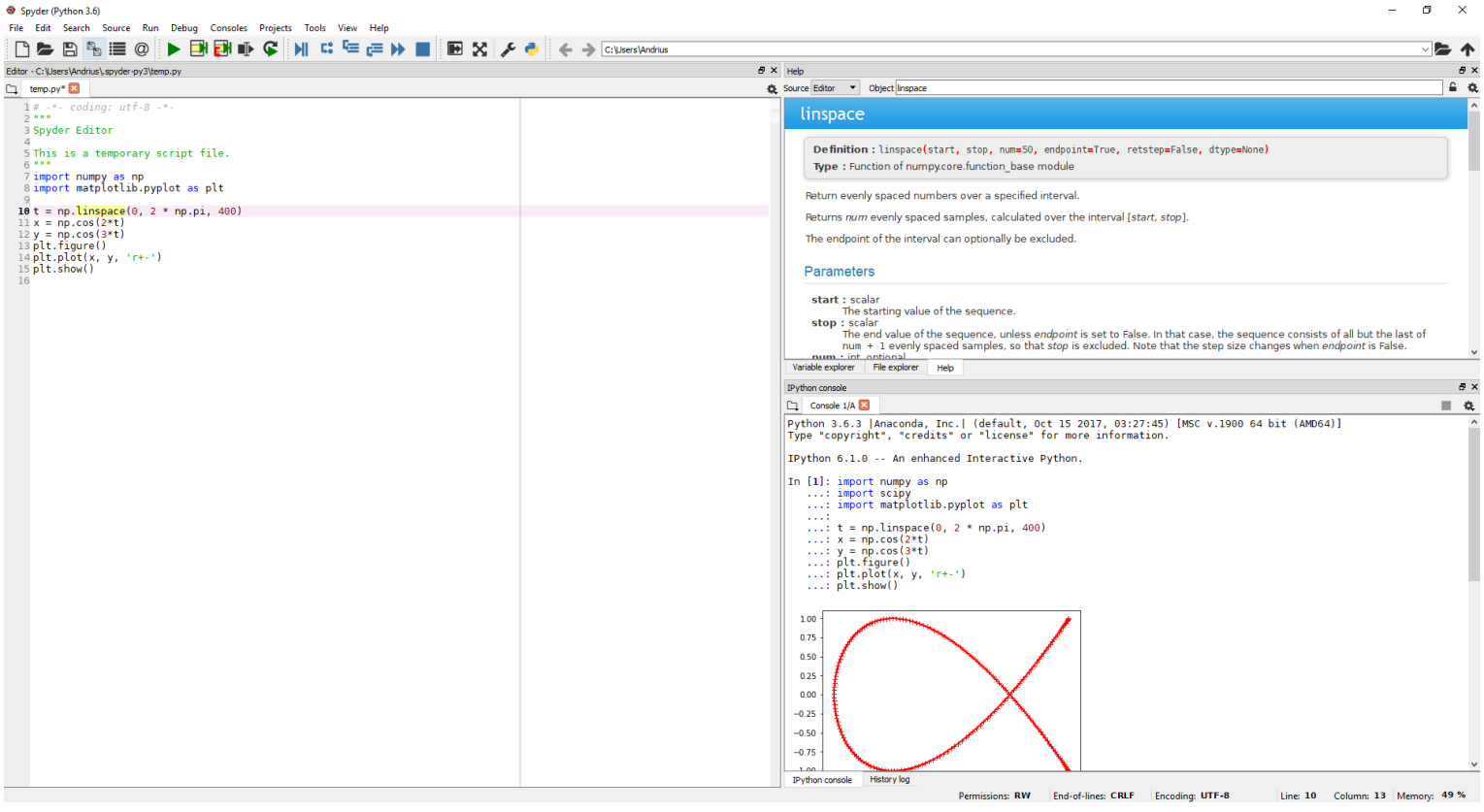


Figure 2: Spyder IDE

Note: the courses will use `JupyterLab`, since it is compatible with both `R` and `Python` and is great for learning. For more focused development `Spyder` may be a great alternative (and in a way similar to `RStudio`).

JupyterLab

Currently in beta, will eventually replace Jupyter Notebook. It allows editing and running notebook documents via a web browser. video

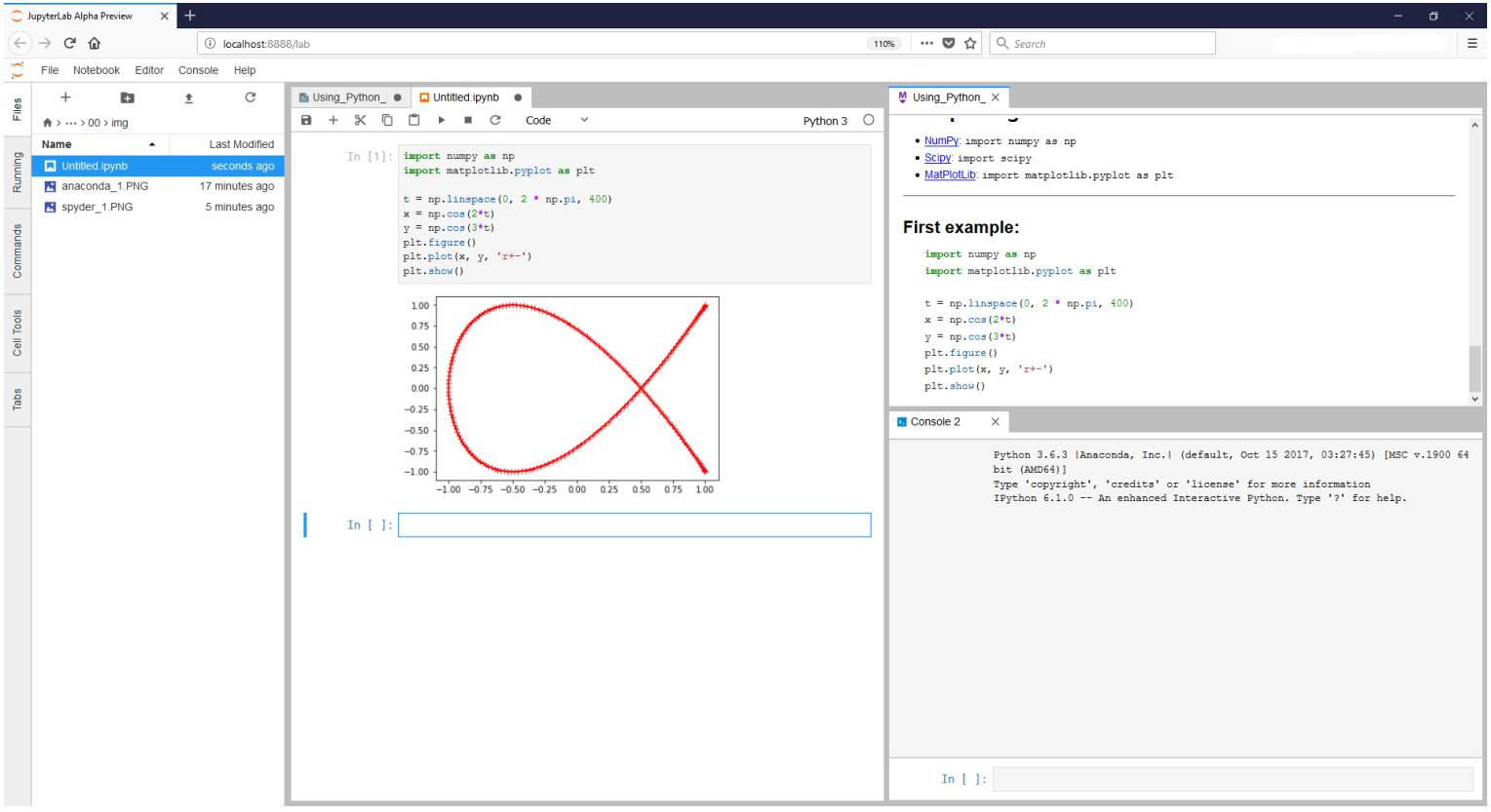


Figure 3: JupyterLab IDE

Importing libraries

Libraries are imported using the `import` command:

- NumPy: `import numpy as np`
- Scipy: `import scipy`
- MatPlotLib: `import matplotlib.pyplot as plt`

Getting Python and R version info:

- in R:

```
print(R.version.string)
```

```
## [1] "R version 3.6.0 (2019-04-26)"
```

- in Python:

```
import sys
py_verion = sys.version
print(py_verion.split("| (", 1)[0])
```

```
## 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916
```

Example with strings:

- in R:

```
first_name = "eric"
print(first_name)
```

```
## [1] "eric"
```

```
print(toupper(first_name))
```

```
## [1] "ERIC"
```

- in Python:

```
first_name = 'eric'
print(first_name)
```

```
## eric
```

```
print(first_name.upper())
```

```
## ERIC
```

A method is something that can be done to a variable. The methods 'lower', 'title', and 'upper' are all functions that have been written into the Python language, which do something to strings.

#Defining functions

- in R:

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

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

- in Python:

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

```
## 5
```

Because Python doesn't use brackets `{...}` when defining functions - Indentation in Python matters! Also in Python:

- If you pass a *mutable* object into a method, the method gets a reference to that **same object**!
- If you pass an *immutable* object to a method, you can't mutate the object.

List - a *mutable* type

- 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("Old: ", old_vec, "\n")
```

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

- in Python:

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

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

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

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

Note - we did not return anything with these functions, the original values were changed in Python!

Workaround:

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: if we write `b_list = a_list` instead of `b_list = a_list[:]` or `b_list = list(a_list)`, then we will again modify the original! However, it is useful to pass by reference instead of value if we do not want our function to return anything but still change the original values.

A list() in R

We can create a list that houses a variety of different data:

```
my_list <- list(type = "my_model", R.sq = 0.99,
                stats = list(values = c(0.3, 0.6, 0.4),
                             evaluated = FALSE))
my_list
```

```
## $type
## [1] "my_model"
##
## $R.sq
## [1] 0.99
##
## $stats
## $stats$values
## [1] 0.3 0.6 0.4
##
## $stats$evaluated
## [1] FALSE
```

A dictionary in Python

```
my_dict = {"type": "my_model",
           "R.sq": 0.99,
           "stats": {"values": [0.3, 0.6, 0.4],
                     "evaluated": True}}
print(my_dict)
```

```
## {'type': 'my_model', 'R.sq': 0.99, 'stats': {'values': [0.3,
```

```
print(type(my_dict))
```

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

```
print(my_dict.keys())
```

```
## dict_keys(['type', 'R.sq', 'stats'])
```

Accessing the values

- in R:

```
my_list$type
```

```
## [1] "my_model"
```

```
my_list$stats$values
```

```
## [1] 0.3 0.6 0.4
```

```
my_list$R.sq * 100
```

```
## [1] 99
```

- in Python:

```
print(my_dict["type"])
```

```
## my_model
```

```
print(my_dict["stats"]["values"])
```

```
## [0.3, 0.6, 0.4]
```

```
print(str(my_dict["R.sq"]*100))
```

```
## 99.0
```

#OLS and result reproducibility example

Generating data from a linear regression model:

- in R:

```
set.seed(123)
nsample  <- 1000
x        <- seq(from = 0, to = 10, length.out = nsample)
x.matrix <- cbind(x, x^2)
beta <- c(1, 0.1, 10)
eps  <- rnorm(nsample)
x.matrix <- cbind(1, x.matrix)
colnames(x.matrix) <- c("const", "x1", "x2")
y    <- x.matrix %*% beta + eps
```

- in Python:

```
#To ommit warning messages from slide output:
import warnings
warnings.filterwarnings('ignore')
```

```
#Import the required modules
import numpy as np
import statsmodels.api as sm
```

```
np.random.seed(123)
nsample  = 1000
x        = np.linspace(0, 10, nsample)
x_matrix = np.column_stack((x, x**2))
beta     = np.array([1, 0.1, 10])
eps      = np.random.normal(size=nsample)
x_matrix = sm.add_constant(x_matrix)
y = np.dot(x_matrix, beta) + eps
```

Estimating an OLS model:

- in R:

```
model <- lm(y ~ x.matrix - 1) #x.matrix has a constant
results <- summary(model)
cat("Parameters: ", results$coefficients[, 1], "\n")
```

```
## Parameters:  1.021681 0.09646229 10.00036
```

```
cat("p-values: ", results$coefficients[, 4], "\n")
```

```
## p-values:  4.331621e-26 0.02650987 0
```

```
cat("R2: ", results$r.squared)
```

```
## R2:  0.9999951
```

- in Python:

```
model = sm.OLS(y, x_matrix)
results = model.fit()
```

You can get the list of the possible attributes with `dir(results)`. For example:

```
print('Parameters: ', results.params)
#print('Standard errors: ', results.bse)
```

```
## Parameters:  [0.94992992 0.12747927 9.99619519]
```

```
print('p-values: ', results.pvalues)
```

```
## p-values:  [1.41044558e-22 3.68521164e-03 0.00000000e+00]
```

```
print('R2: ', results.rsquared)
```

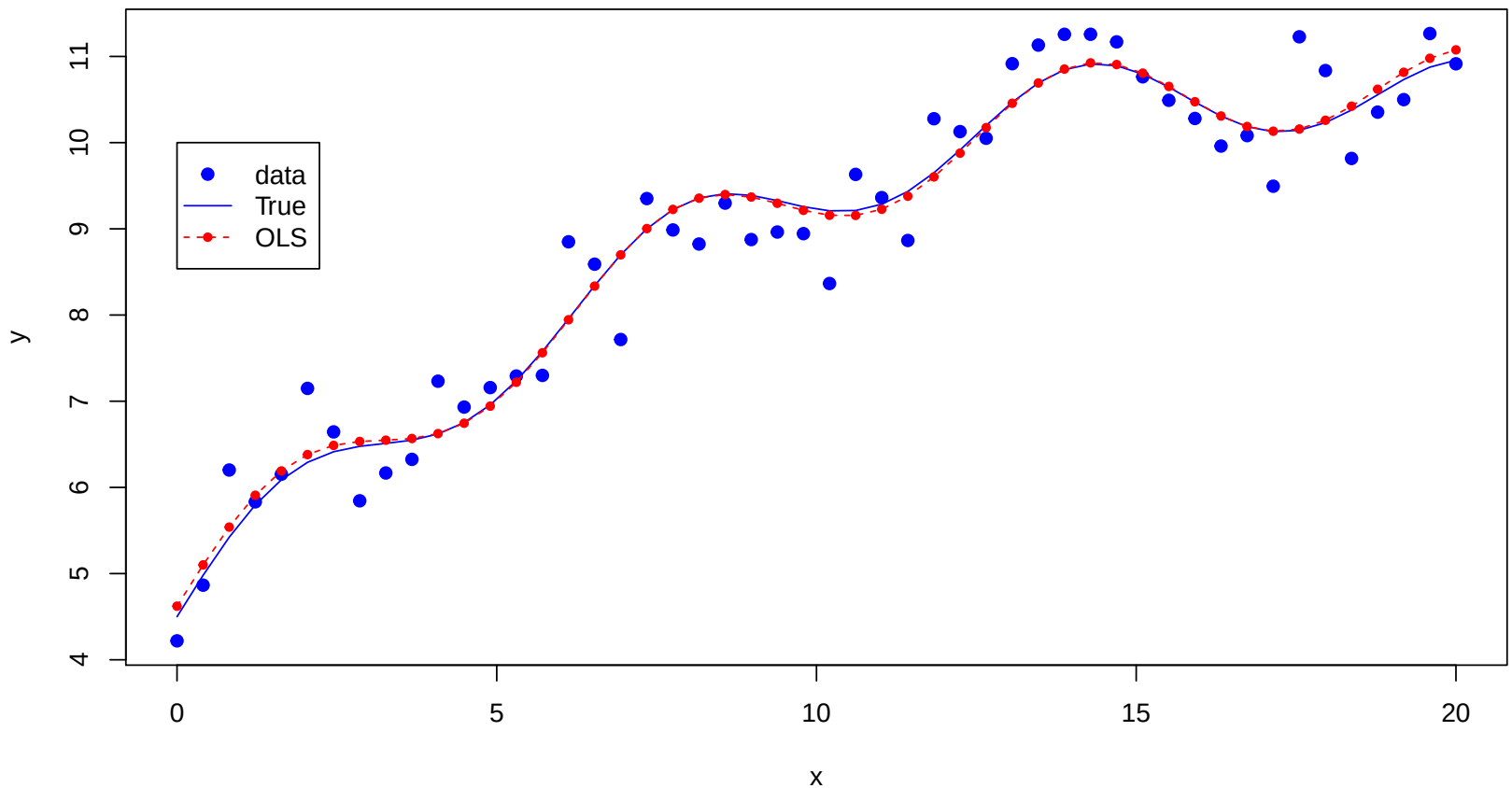
```
## R2:  0.9999887941148841
```

#OLS Example No.2

- in R:

```r
#Simulation:
set.seed(123)
nsample  <- 50
sig      <- 0.5
x        <- seq(from = 0, to = 20, length.out = nsample)
x.matrix <- cbind(1, x, sin(x), (x - 5) ^ 2)
beta <- c(5, 0.5, 0.5, -0.02)
eps  <- rnorm(nsample)
y.true <- x.matrix %*% beta
y      <- y.true + sig * eps
#Estimation:
y.mdl <- lm(y ~ x.matrix - 1)
```

```r
plot(x, y, type = "p", col = "blue", pch = 19)
lines(x, y.true, col = "blue", lty = 1, lwd = 1)
lines(x, y.mdl$fitted.values, col = "red", lty = 2,
      type = "o", pch = 20)
legend(0, 10, c("data", "True", "OLS"),
       col = c("blue", "blue", "red"),
       lty = c(NA, 1, 2), pch = c(19, NA, 20))
```



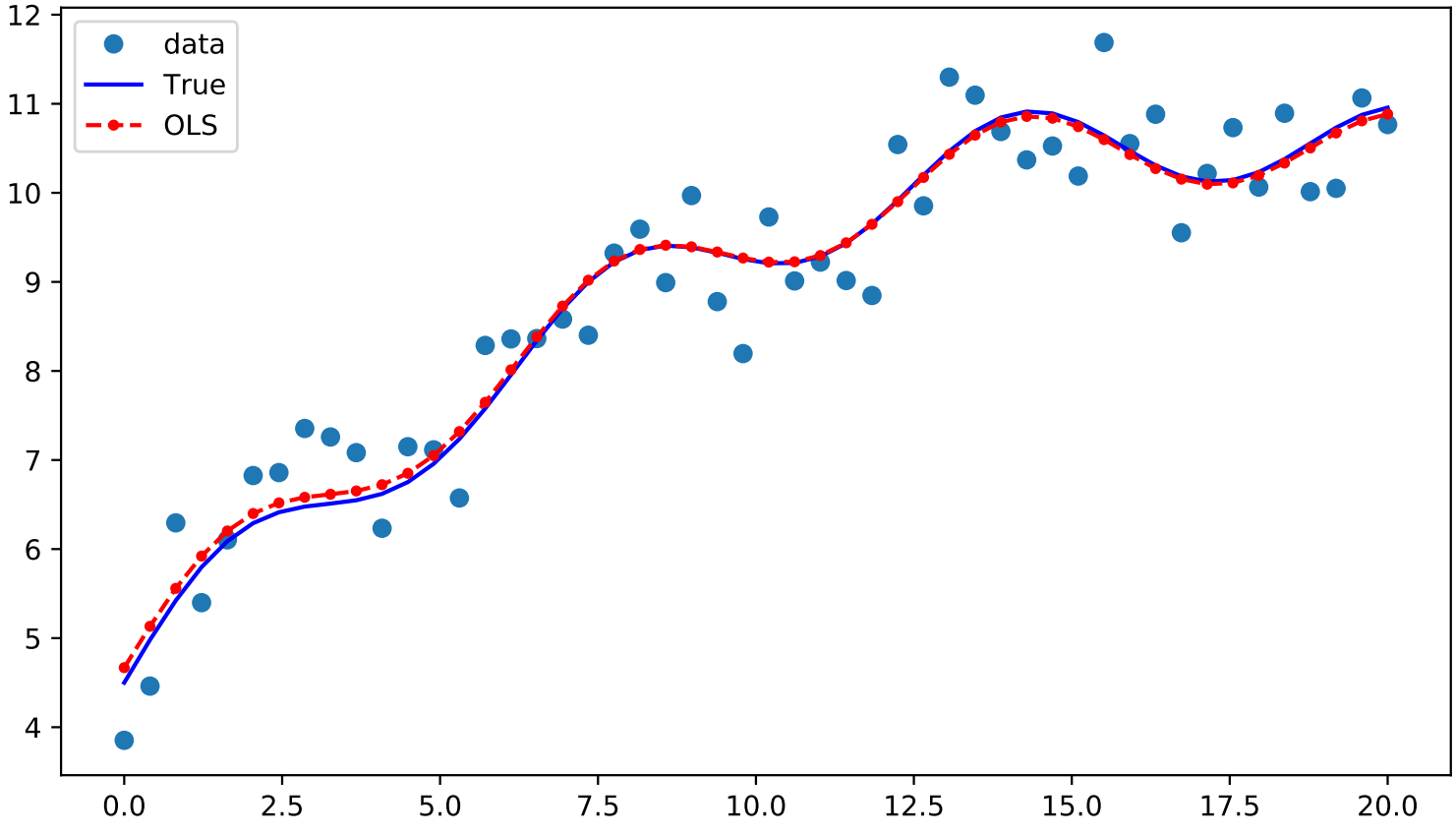
- in Python:

```
import numpy as np
import statsmodels.api as sm
```

```
#Simulation:
np.random.seed(123)
nsample  = 50
sig      = 0.5
x        = np.linspace(0, 20, nsample)
x_matrix = np.column_stack((np.ones(nsample), x,
            np.sin(x), (x-5)**2))
beta   = np.array([5., 0.5, 0.5, -0.02])
eps    = np.random.normal(size=nsample)
y_true = np.dot(x_matrix, beta)
y =  y_true + sig * np.random.normal(size=nsample)
#Estimation:
y_mdl = sm.OLS(y, x_matrix).fit()
```

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots(figsize=(9,5))
ax.plot(x, y, 'o', label="data")
ax.plot(x, y_true, 'b-', label="True")
ax.plot(x, y_mdl.fittedvalues, 'r--.', label="OLS")
ax.legend(loc='best')
```



Classes in Python

Classes allow you to define the information and behavior that characterize anything you want to model in your program.

We will define a class to estimate an OLS regression:

- Our class will require an endogeneous variable y and a matrix X of exogenous variables (with a constant) to be specified;
- Our class will estimate:
  - model coefficients;
  - coefficient standard errors;
  - coefficient significance t-statistic & p-values;
- we will compare the results with the ones from the 'statsmodels.api'.

The following libraries are used:

```
import numpy as np
from scipy import stats
```

Note: **methods** are just **functions** which are defined for a class.

```
import numpy as np
from scipy import stats

class CustomOLS():

    #Method that sets the values for any parameters that need to be defined when an object is first created
    #The "self.*" part is a syntax that allows access to a variable from anywhere else in the class
    def __init__(self, y, x_matrix):
        self.y = y
        self.data = x_matrix
        self.params = None

    #Our estimation method
    def fit(self):
        #Estimate the coefficients:
        x_t = np.transpose(self.data)
        x_prod = np.dot(x_t, self.data)
        x_inv = np.linalg.inv(x_prod)
        self.params = np.dot(x_inv, np.dot(x_t, self.y))
        #Calculate std. err & t-statistic:
        MSE = sum((self.y - self.calcFitted())**2) / (len(self.data) - len(self.data[0]))
        var_b = MSE * x_inv.diagonal()
        self.std_e = np.sqrt(var_b)
        self.t_stat = (self.params - 0) / self.std_e
        #Calculate the p-values:
        self.p_val = []
        for i in self.t_stat:
          temp_val = stats.t.cdf(np.abs(i), (len(self.data)-1))
          self.p_val.append(2 * (1 - temp_val))

    #Method to calculate the fitted values
    def calcFitted(self):
        if self.params is None:
          print("Cannot calculate fitted values - parameters not estimated!")
          return None
        fittedvalues = np.dot(self.data, self.params)
        return fittedvalues
```

```
my_ols = CustomOLS(y, x_matrix)
my_ols.calcFitted()
```

```
## Cannot calculate fitted values - parameters not estimated!
```

```
my_ols.fit()
print(np.round(my_ols.params, 4))
```

```
## [ 5.1548  0.4834  0.4807 -0.0194]
```

```
print(np.round(my_ols.std_e, 4))
```

```
## [0.1874 0.0289 0.1136 0.0025]
```

```
print(np.round(my_ols.t_stat, 4))
```

```
## [27.5076 16.7244  4.2309 -7.6644]
```

```
print(np.round(my_ols.p_val, 4))
#my_ols.calcFitted()
```

```
## [0.     0.     0.0001 0.    ]
```

Compared to the output from `statsmodels.api`:

```
print(np.round(y_mdl.params, 4))
```

```
## [ 5.1548  0.4834  0.4807 -0.0194]
```

```
print(np.round(y_mdl.bse, 4))
```

```
## [0.1874 0.0289 0.1136 0.0025]
```

```
print(np.round(y_mdl.tvalues, 4))
```

```
## [27.5076 16.7244  4.2309 -7.6644]
```

```
print(np.round(y_mdl.pvalues, 4))
```

```
## [0.     0.     0.0001 0.    ]
```

- classes are usually saved in a separate file, and then imported into the program you are working on.
- This allows you to write "reusable" code which can be applied to different projects, saving development time.

For example, save the `CustomOLS` class in a `custom_ols.py` file. To import you class use:

```
import custom_ols as my
```

In case you get an error `ModuleNotFoundError: No module named 'custom_ols'`, use the following code to add you current working directory (make sure that it is the same one that your current code `.py` file is located!) to the Python consoles list of paths:

```
#Get current working directory
import os
cwd = os.getcwd()

#Add the current working directory to the list of Python system paths:
import sys
sys.path.append(cwd)

#Import your custom class:
import custom_ols as my

#Evaluate your model:
my_OLS_class = my.CustomOLS(y, x_matrix)
my_OLS_class.calcFitted()
my_OLS_class.fit()
my_OLS_class.params
```

Create a list of OLS models in Python

```
my_models = []
for i in range(0, len(x_matrix[0])):
  #Note: use my.CustomOLS if continuing from previous slide
  my_models.append(CustomOLS(y, x_matrix[:, range(0, i+1)]))
  my_models[i].fit()
  print("coef: " + str(np.round(my_models[i].params, 4)))
  print("pval: " + str(np.round(my_models[i].p_val, 4)))
```

```
## coef: [8.8455]
## pval: [0.]
## coef: [6.0321 0.2813]
## pval: [0. 0.]
## coef: [5.9648 0.2868 0.3464]
## pval: [0.    0.    0.044]
## coef: [ 5.1548  0.4834  0.4807 -0.0194]
## pval: [0.     0.     0.0001 0.    ]
```

Note: You can use `*.py` files to store a set of functions you want available in different programs as well, even if those functions are not attached to any one class. Example:

```
#save as custom_sum.py
def my_add(x, y):
  x = x + 1
  y = y + 1
  return x + y
```

Then load the module in our main code file (as before, make sure that the Python path list has the directory of the `custom_sum.py` file):

```
import custom_sum as m
print(str(m.my_add(1, 2)))
```