EEMP_python_intro

October 16, 2019

1 EEMP - Introduction to Python for Data Analysis

- Introductory course to working with python for data analysis
- Goals:
  - Overview of basic data structures and commands in python
  - Essential toolkit for data analysis in python, giving you a background in packages needed
  - By no means exhaustive, but should enable you to continue learning by yourself
- More commands will be introduced and practiced throughout the course

2 Organizational issues

- 3 python introductory sessions:
  - 07/10/2019, 14:00 - 15:30
  - 07/10/2019, 16:00 - 17:30
  - 08/10/2019, 10:00 - 11:30
- all materials can be found in the course github:

  https://jeshan49.github.io/EEMP2019/
- We will be working with mainly three tools within this course
  - Python (version 3.7.3)
  - Jupyter notebook
  - Spyder

Let's set the scene for working with python and jupyter notebook:

Step 1: install a python distribution and the respective packages

- we will be using Anaconda https://www.anaconda.com/
- install the required packages (in this order): numpy, pandas, statsmodels, matplotlib, seaborn, scikit-learn - open Anaconda/Environments -> check whether respective packages already installed, if not install - command line: conda install -c anaconda numpy pandas statsmodels seaborn scikit-learn

```
.. check installed packages with:
conda list
```

Step 2: open a jupyter notebook

- with Anaconda

- command line: jupyter notebook

3 8 Reasons Why You Should Learn Python

1. Consistently ranks among the most popular programming languages with a promising future
2. First-class tool for scientific computing tasks, especially for large datasets
3. Straightforward sytnax and easy to learn
4. Very versatile and highly compatible
5. Free of charge since it is open source
6. Comprehensive standard libraries with large ecosystem of third-party packages
7. State-of-the-art for machine learning and data science in general
8. Great amount of resources and welcoming community

Let's get started with Python...

3.1 1. Datatypes and Operators

3.1.1 Datatypes:

- Integers
- Floats (decimal number)
- Strings ("text")
- Booleans (TRUE/FALSE)

[ ]:
```
# integers
a = 10
b = 4

print(a)
print(type(a))
(a+b)*3
```

[ ]:
```
# want to know more about a function and how to use it? - use the help()␣
↪function or internet search
help(print)
help(type)
```

[ ]:
```
# floats
c = 1.5
d = 28.0

print(type(d))
c*d
```

[ ]:
```
# strings

question = "What is the answer to life, to the universe and everything?" #␣
↪either denote with ""
```

```
answer = '42' # .. or ''

print(type(question))

question_answer= question + answer # strings can be added too!

print(question_answer)

print(question," - ",answer)
print(question + ' - ' + answer)
```

[ ]:
```
# Booleans and True/False - Operators


print(True==1) # True is encoded as 1
print(False==1) # False is encoded as 0

print(not True) # we can inverse a boolean with "not"
print(True + True) # We can also add booleans
```

[ ]:
```
# we can evaluate the truth of a statement with the different operators ==,!
 ↪=,>,<,>=,<=, no
# -> the output is always a boolean

print(True > False)
print(answer == '42')
print(4 >= 5)
print(10 != 0)
```

[ ]:
```
# we can also combine different conditions with and, &, or

print(2>1 and 2<3)
print(1==1 & 2==2)
print(2==1 or 2<3)
```

[ ]:
```
# If-statements can be used to execute code only if a certain condition is␣
 ↪fulfilled

answer = '42'

# identation after if-condition needed (convention is to indent with 4 spaces)
if answer == "42":
    print("This is the answer to life, to the universe and everything.")
```

[ ]:
```
# we can also include additional conditions

answer = '5'
```

```
if answer == "42":
    print("This is the answer to life, to the universe and everything.")
elif answer == "41" or answer == '43':
    print("This is nearly the answer to life, to the universe and everything.")
else:
    print("This is not the answer to life, to the universe and everything.")
```

3.2 2. Python Lists

- Standard mutable multi-element container in Python
- Denoted by squared brackets [ ]

[ ]:
```
# Python lists can contain integers...

l1 = list(range(10))
print(l1, type(l1[0]))

# ...strings

l2 = list(str(i) for i in l1)
print(l2, type(l2[0]))


# ... or a combination of different data types.

l3 = [1.2,42,'Yes',True]
print(l3)
print([type(i) for i in l3])
```

[ ]:
```
# one can also access the different elements within a list with calling its␣
↪index

print(l1[0]) # Python is a zero-indexed language, i.e. this would give you the␣
↪first element of the list
print(l2[-1]) # one can also access the list from the end, i.e. this would give␣
↪you the last element of the list
print(l2[-2]) # ... this the second last element

print(l3[0:3]) # or slice the list and extract only a certain range
```

3.3 3. Loops

- We can loop over values in a list

[ ]:
```
for item in ['life','the universe','everything']:
    print('The answer to',item,'is 42.')
```

[ ]:
```
even_number = list(range(0,10,2)) # check help(range) to find out about the␣
↪options within the function

print(even_number)

result = 0
for number in even_number:
    result += number # this is the inplace version of reassigning result =␣
↪result + number, the outcome is identical
    print(result)
```

3.4 4. Functions

- Python is also suitable for writing functions
- Very good for operations that are done repeatedly, but have no built-in functions
- However, whenever there are built-in functions always use those; they are usually computationally more efficient
- We will give you a short idea of what a function means and how it looks like, but writing functions is not the focus of the course

[ ]:
```
def f(x):
    '''This function squares its numerical input'''
    return x**2
```

[ ]:
```
f(3)
```

3.5 5. Libraries

3.5.1 5.1 NumPy Library

Provides numeric vector and matrix operations

- NumPy's "ndarray" is another of the basic formats data can be stored in
- Similar to python built-in lists (see 2.), but lack its multi-type flexibility, i.e. can only contain one data type
- However, in contrast to lists, ndarrays are more efficient in storing and manipulating data, which is important as data become bigger
- Building blocks for many other packages (see 5.2)

[ ]:
```
# Before we can use a package the first time, we need to import it (given we␣
↪have it already installed)
# the "as np" indicates the alias we can use to call the package from now on
import numpy as np
```

[ ]:
```
# ndarrays
array1 = np.array([0,1,5,15,2])
print(array1, type(array1))
```

```
array2 = np.arange(5)
print(array2)

array3 = array1 + array2 # ndarrays can also be added to each other
print(array3)
```

[ ]:
```
# we can also build matrices from ndarrays

matrix1 = np.array([[1,0],[0,1]])
print(matrix1, type(matrix1))

matrix2 = np.array([array1, array2, array1 + array3])
print(matrix2)

matrix3 = matrix2 + array1
print(matrix3)
```

[ ]:
```
# and then work with these arrays and matrices using numpy methods and functions

matrix2_t = matrix2.transpose()
print(matrix2_t)
print(np.shape(matrix2_t)) # gives you a 5x3 matrix from the original 3x5 matrix
```

[ ]:
```
# as with lists you can access elements within an array in a similar fashion

print(array1[0:4:2]) # slicing scheme: array[start:stop:step]

print(matrix2_t[1,2]) # takes only the index 1 row- and index 2 column-element
print(matrix2_t[0:2,0:2], np.shape(matrix2_t[0:2,0:2])) # gives you a 2x2␣
↪matrix from the 5x3 original matrix
```

3.5.2 5.2 Pandas Library

Provides the DataFrame, which is the building block for working with data

- Newer package built on top of NumPy
- A Series is the Pandas equivalent to a NumPy array
  - However, Series (and DataFrames) have explicitly defined row (and column) labels attached as compared to implicitly defined simple integer indices in NumPy
- A DataFrame is a 2-dimensional multi-array data structure, i.e. consist of multiple series
- Often contain heterogenous types and/or missing values

[ ]:
```
# Again, we have to import the package first...
import pandas as pd
```

[ ]:
```
# A series is a one-dimensional array of indexed data

series1 = pd.Series([0,2,4,6])
```

```
print(series1)

print(series1.values) # can access the values within a Series

print(series1.index) # can access the respective indices of a Series
```

[ ]:
```
# However, indices don't need to be numerical, but could also be strings...

series2 = pd.Series([1,3,6], index=['a','b','c'])

print(series2)
```

[ ]:
```
# Now, let's read in an actual dataset with several columns and numerous rows,␣
↪and start working with DataFrames...

path_to_data = "https://raw.githubusercontent.com/lemepe/EEMP/master/
↪python_intro/Employee_data.csv"

employee_data = pd.read_csv(path_to_data)

# We can inspect the data by looking at the first few rows

employee_data.head() # by default this gives you the first 5 observations in␣
↪the dataframe
```

3.5.3 Commands for exploratory data analysis (EDA)

[ ]:
```
# Shape of the dataframe in form of a tuple (#rows, #cols)

employee_data.shape
```

[ ]:
```
# Lists all column indeces

employee_data.columns
```

[ ]:
```
# Overview of columns, non-null entries & datatypes

employee_data.info()
```

[ ]:
```
# Summary statistics of the dataset

pd.set_option('display.max_columns', 200) # this command sets the number of␣
↪displayes columns to 200

employee_data.describe()
```

```
[ ]: # To get an idea about the different values contained within a specific column,␣
     ↪we can use the .unique() method
     # since unique takes the values in order of appearance, we use the sorted␣
     ↪function on top of it
     # with [] and the respective column label, we can access this particular Series␣
     ↪in the DataFrame

     print(sorted(employee_data['DistanceFromHome'].unique()))

     print(sorted(employee_data['WorkLifeBalance'].unique()))
```

```
[ ]: # If we want to know more about the distribution of certain values or␣
     ↪categories, we can use value_counts()

     print(employee_data['WorkLifeBalance'].value_counts()) # this would give the␣
     ↪frequency counts in descending order

     print(employee_data['WorkLifeBalance'].value_counts(normalize=True)) #␣
     ↪alternatively, we can show percentages
```

```
[ ]: # We can also slice the data by indices (similar to how we did it with lists or␣
     ↪NumPy arrays)

     employee_data.loc[0:100] # extract first 100 observations by referring to the␣
     ↪explicitly defined index ...
```

```
[ ]: # .. which could also be a string, as in the case of the columns

     employee_data.loc[0:100,['MonthlyIncome','Department']]
```

```
[ ]: #  In contrast to iloc, which uses the implicitly defined row and column index

     employee_data.iloc[0:100,[4,18]] # sliced both row- and columnwise by implicit␣
     ↪indeces
```

```
[ ]: # select subset of data with a condition and assign it to new dataframe

     exit_data = employee_data[employee_data['Attrition']=='Yes']

     # we can also subselect only certain columns to be shown

     print(exit_data[['MonthlyIncome','Department']].head(10))

     # ... or only select rows that fulfill a certain condition

     age_mon_inc = employee_data.loc[:,['Age','MonthlyIncome']]
```

```
print(age_mon_inc)

u35_mon_inc = employee_data.loc[employee_data['Age']<35,['Age','MonthlyIncome']]
print(u35_mon_inc)
```

3.5.4 Descriptives statistics

- Overview of pandas aggregation methods:

[ ]:
```
# Distribution of exits across departments
print(exit_data['Department'].value_counts(normalize=True))

# Mean age of exits
mean_age_exited = exit_data['Age'].mean()
print(mean_age_exited)

# Mean age of exits across departments
mean_age_exited_by_dep = exit_data['Age'].groupby(exit_data['Department']).
↪mean() # groupby allows for groupwise calculations
print(mean_age_exited_by_dep)

# Mean age across all employees
mean_age = employee_data['Age'].mean()
print(mean_age)
```

[ ]:
```
# Creating new variables

employee_data['dummy_exit'] = 0 # create column 'dummy_exit' with only zeros
employee_data.loc[employee_data['Attrition'] =='Yes', 'dummy_exit'] = 1 #␣
↪replace with one if 'Attrition'==Yes

employee_data['dummy_u35'] = employee_data['Age']<35 # Dummy with boolean when␣
↪attrition equal to 'Yes'

# Alternatively pandas has an already integrated command to create dummies from␣
↪categorial variables

employee_data=pd.
↪get_dummies(employee_data,columns=['BusinessTravel','EducationField'])

employee_data.head()
```

[ ]:
```
# calculate percentage of employees that left the company

employee_data['dummy_exit'].sum()/len(employee_data) # len() gives the length,␣
↪i.e. number of rows of an array or df
```

```
[ ]: # alternatively using value_counts()

     employee_data['dummy_exit'].value_counts(normalize=True)
```

```
[ ]: # with the groupby() function, we can split the dataset by categories and do␣
      ↪calculations on these subgroups

     employee_data['dummy_exit'].groupby(employee_data['Department']).
      ↪value_counts(normalize=True)
```

3.5.5 5.3 Visualization Libraries

Provide plotting and visualization support

Matplotlib Library

- Original visualization library built on NumPy arrays
- Conceived in 2002 to enable MATLAB-style plotting
- We will only provide a quick overview, for more information see matplotlib documentation
- https://matplotlib.org/3.1.1/gallery/index.html

```
[ ]: import matplotlib.pyplot as plt
```

```
[ ]: # Here we define the plotstyle to be used
     # check https://matplotlib.org/3.1.1/gallery/style_sheets/
      ↪style_sheets_reference.html for an overview of style sheets

     plt.style.use('ggplot')
```

```
[ ]: # There exist several "magic functions" in jupyter notebook which allow you␣
      ↪additional operations
     %lsmagic
```

```
[ ]: # ... one we will need is "%matplotlib inline" which enables matplotlib plots␣
      ↪to be displayed directly in the notebook
     %matplotlib inline
```

```
[ ]: # Histogram with frequencies
     plt.hist(employee_data['Age'])
     plt.xlabel('Age')
```

```
[ ]: plt.hist(employee_data['Age'],density=True, alpha=0.5)
     plt.hist(exit_data['Age'],density=True, alpha=0.5)
     plt.xlabel('Age')
```

[ ]:
```
# We can also combine multiple plots with plt.subplot(#rows,#cols,i)

plt.subplot(1,2,1)
plt.hist(employee_data['Age'],density=True, color = 'red', alpha=0.5)
plt.xlabel('Age (all)')
plt.xlim(18,60)
plt.ylim(0,0.06)

plt.subplot(1,2,2)
plt.hist(exit_data['Age'],density=True, color = 'blue', alpha=0.5)
plt.xlabel('Age (exits)')
plt.xlim(18,60)
plt.ylim(0,0.06)
```

[ ]:
```
# Scatter plot example
plt.scatter(employee_data['YearsAtCompany'],employee_data['MonthlyIncome'])
plt.xlim(0,)
plt.xlabel('YearsAtCompany')
plt.ylabel('Monthly income')
```

[ ]:
```
plt.subplot(2,2,1)
plt.
 ↪scatter(employee_data[employee_data['Education']==2]['YearsAtCompany'],employee_data[employe
plt.xlim(0,50)
plt.xlabel('Years at Company')
plt.ylabel('Monthly income')

plt.subplot(2,2,2)
plt.
 ↪scatter(employee_data[employee_data['Education']==3]['YearsAtCompany'],employee_data[employe
plt.xlim(0,50)
plt.xlabel('Years at Company')
plt.ylabel('Monthly income')

plt.subplot(2,2,3)
plt.
 ↪scatter(employee_data[employee_data['Education']==4]['YearsAtCompany'],employee_data[employe
plt.xlim(0,50)
plt.xlabel('Years at Company')
plt.ylabel('Monthly income')

plt.subplot(2,2,4)
plt.
 ↪scatter(employee_data[employee_data['Education']==5]['YearsAtCompany'],employee_data[employe
plt.xlim(0,50)
plt.xlabel('Years at Company')
plt.ylabel('Monthly income')
```

```
plt.subplots_adjust(hspace=0.5,wspace=0.5)
plt.show()
```

Seaborn Library

- Newer library with more visually appealing and simpler to use toolkit
- Better suited for visualizing DataFrame structures
- Again, we only provide a quick overview, see documentation for more details
- https://seaborn.pydata.org/examples/index.html

[ ]:
```
import seaborn as sns

sns.set()
```

[ ]:
```
sns.distplot(employee_data['MonthlyIncome'],axlabel='Years at Company')
```

[ ]:
```
sns.
 ↪distplot(employee_data[employee_data['Gender']=='Male']['YearsAtCompany'],axlabel='Years␣
 ↪at company')
sns.
 ↪distplot(employee_data[employee_data['Gender']=='Female']['YearsAtCompany'],axlabel='Years␣
 ↪at company')
```

[ ]:
```
sns.relplot(x='YearsAtCompany',y='MonthlyIncome',data=employee_data,␣
 ↪hue='Gender')
```

[ ]:
```
sns.
 ↪regplot(x='YearsAtCompany',y='MonthlyIncome',data=employee_data[employee_data['Gender']=='M
sns.
 ↪regplot(x='YearsAtCompany',y='MonthlyIncome',data=employee_data[employee_data['Gender']=='F
```

[ ]:
```
sns.barplot(x='JobLevel',y='MonthlyIncome',data=employee_data)
```

3.5.6 5.4 Statsmodels Library

Provides many different statistical models, statistical tests, and statistical data exploration

- https://www.statsmodels.org/stable/index.html

[ ]:
```
# import the statsmodels library
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

[ ]:
```
# simple OLS regression with one explanatory variable
```

```
results_ols = smf.ols('MonthlyIncome ~ YearsAtCompany', data = employee_data).
 ↪fit()
print(results_ols.summary())
```

[ ]:
```
# OLS regression with several explanatory variables
results_ols2 = smf.ols('MonthlyIncome ~ YearsAtCompany + C(JobLevel) +␣
 ↪C(Gender)+ C(Department)', data = employee_data).fit()
print(results_ols2.summary())
```

[ ]:
```
# Logit regression with one explanatory variable
results_logit = smf.logit('dummy_exit ~ MonthlyIncome', data = employee_data).
 ↪fit()

print(results_logit.summary())
```

[ ]:
```
# Logit regression with several explanatory variables
results_logit2 = smf.logit('dummy_exit ~ MonthlyIncome + Age + C(JobLevel) +␣
 ↪C(WorkLifeBalance) + C(JobSatisfaction) + TrainingTimesLastYear', data =␣
 ↪employee_data).fit()

print(results_logit2.summary())
```

[ ]:
```
# Multiple regressions in one table - "paper format"

from statsmodels.iolib.summary2 import summary_col

print(summary_col([results_logit,results_logit2],stars=True))
```

3.5.7 5.5 Scikit-learn Library

Provides general purpose machine learning package with extensive coverage of models and feature transformers

- not part of the introductory session, but will be introduced in a later part of the course

4 6. References and Further Readings

- VanderPlas, Jake (2016). Python Data Science Handbook: Essential Tools for Working with Data. O'Reilly Media.
- McKinney, Wes (2012). Python for Data Analysis. O'Reilly Media.
- Rule et al. (2018). "Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks". In: PLoS Computational Biology 15.7: e1007007.
- Athey, Susan and Guido W. Imbens (2019). "Machine Learning Methods That Economists Should Know About". In: Annual Review of Economics 11, pp. 685-725.

- Breiman, Leo (2001). “Statistical Modeling: The Two Culutures”. In: Statistical Science 16.3, pp.199-231.
- Hastie, Trevor, Tibshirani, Robert, and Jerome Friedman (2017). Elements of Statistical Learning. Springer Science & Business Media.
- Mullainathan, Sendhil and Jann Spiess (2017). “Machine Learning: An Applied Econometric Approach”. In: Journal of Economic Perspectives 31.2, pp. 87-106.- VanderPlas, Jake (2016): Python Data Science Handbook: Essential Tools for Working with Data. O’Reilly Media.
- Adams, Douglas (2008): The Hitchhiker’s Guide to the Galaxy. Reclam.