***** Introduction to Python for ML *****

Abdelkrim El mouatasim
Full Professeur of Applied Mathematics - AI

UIZ FPO - DMG

2022/2023

Find all slides on `https://sites.google.com/a/uiz.ac.ma/elmouatasim/` (Master-MASED: Machine Learning

Agenda for this Unit

Introduction **5**

- What is Python? 5
- Installation and Components 7

Variables and Data Types **9**

- Basic Data Types 9
- Sequence Data Types 13
- Mapping Types 18

Logical Operators **20**

Logical Values 20

Control Flow 22

Conditionals 22

Loops 23

Python OO 25

What are Classes? 25

Class Variables vs. Instance Variables 27

Inheritance 29

Machine Learning Libraries 30

NumPy 30

ScikitLearn 37

Matplotlib (PyPlot) 40

Introduction

What is Python?

- Python was developed by Guido van Rossum in the early 1990s.
- It is an interpreted, high-level, general-purpose programming language which is the state-of-the-art for data science and machine learning.
- **White space has a meaning!** (important difference to other languages like Java or C++)
  - Semantic code blocks (e. g. conditions, loops, ...) have to be indented.
  - You may use tabs or spaces for indentation, but the interpreter **does not allow you to mix these two!**
- Python supports many programming paradigms: We can write **procedural**, **functional** or **object-oriented** programs.

- **Caveat:** Python has two **non-compatible** branches:
  - `Python 2.7` $\Longleftarrow$ This one is deprecated and no longer maintained
  - `Python 3.x` $\Longleftarrow$ Use this one!

Installation and Components

Python

- Python is pre-installed on Mac and Linux. You may check this by executing `python -V` in the command line.
- For Windows (and Mac), download the newst version from ⇒ www.python.org/downloads/.
- Don't forget to check the '`Add Python 3.x to PATH`' option!

Jupyter

- It is a browser-based Python environment (either on local or remote machine).
- The code is editable directly in the browser.
- Nice for (interactive) demonstrations, **but hard to debug**.

Anaconda

- A package manager for creating **isolated Python environments** (with different package versions)
- It is good programming practice to create separate environments for different projects in order to avoid version mismatches, package clashes etc.
- It can be downloaded from ⇒ https://www.anaconda.com/download/

I recommend to use the *Anaconda Navigator* which combines all components, including a Python IDE called *spyder*.

Variables and Data Types

Basic Data Types

- Unlike Java, Python is **typed dynamically**, i. e. the data type does not have to be specified when declaring variables.
- Nevertheless, Python works with these types internally (cf. ⇒ table 2).

Type	Description	Example
`int`	Integer (no maximum for integers, only limited by memory size)	0, 1, 2, ...
`float`	Floating point number	3.1415, 2.7182
`str`	String (unicode characters)	'Hello world'
`bool`	Boolean values	`True` or `False`

Table 1: Basic data types in Python

Some examples

```python
# This declares a variable s initialized with value "Hello"
s = 'Hello'

# You may also use double quotes
s = "Hello"

# Indexing is possible (starts at 0)
s = "Hello"[1] # gives "e"

# Indexing from the back
s = "Hello"[-1] # gives "o"

# Dynamic typing: Now you can use the same variable to save numbers,
# but keep track of what you do with your variables!
s = 0.50**2 * 3.1415
```

Conversion between Types

- Although you don't have to bother with types in Python, you may want to convert a variable to another type.
- E. g. when concatenating a number to a string

```
# Conversion between types
int("42") # gives 42
str(42) # gives "42"

# Example:
# This does NOT work (TypeError: can only concatenate str to str)
print("The answer is " + 42)

# Use type conversion:
print("The answer is " + str(42))
# Or alternatively:
print("The answer is", 42)
```

Immutable data types

```python
# Example for int data type:
a = 10
id(a)        # gives e.g. 1838375360

a = a + 1
id(a)        # gives e.g. 1838375392

# Immutability means that some data types cannot be modified:

# ------------------------
# This code does NOT work!
# ------------------------
var = "Hello"
var[1] = "E"
```

Sequence Data Types

- Sometimes it is necessary to store multiple elements in one place. Think e. g. of a list of data points which we want to analyze using machine learning methods.
- Python offers the types `range`, `list` and `tuple` for that.

Ranges

- The `range` type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for-loops.
- **Interface:** class `range`(*start*, *stop*[, *step*])
  - *start*: Fist point in the sequence of numbers (default = 0).
  - *stop*: Last point in the sequence of numbers (excluding).
  - *step*: Step size

Lists

- Lists are **mutable** sequences of elements (i. e. elements can be inserted, modified, deleted).
- Lists can be constructed in several ways:
  - Using a pair of square brackets (empty list): `[]`
  - Using square brackets, separating items with commas: `[a]`, `[a, b, c]`
  - Using the type constructor: `list()` or `list(iterable)`
  - Using a **list comprehension**: `[x for x in iterable]`
- Example: `list(range(1, 10)) = [1, 2, 3, 4, 5, 6, 7, 8, 9]`

Lists are commonly used to store *homogeneous* types of elements, e. g. a list of customers.

- List comprehensions originate from **functional programming** and provide a concise way to create lists.
- Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.
- Example: Compute square numbers:

```python
# Conventional way:
squares = []
for x in range(10):
    squares.append(x**2) # add element to the list

# Result: [0, 1, 4, 9, 16, 25, ...]

# List comprehensions:
squares = [x**2 for x in range(10)]
```

Task: Create a list of prime numbers up to n by using list comprehensions only!

Excursion: Sets

- Python also includes a data type for sets.
- Following the mathematical notion of a set, it is an **unordered** collection with **no duplicate** elements.
- A set can be created using `set()` (not `{}`!)
- `set([4, 1, 2, 2, 6])` gives `{ 1, 2, 4, 6 }`
- `set([4, 1, 2, 2, 6]) == set([4, 4, 4, 1, 2, 6, 2])` evaluates to `True`!

Tuples

- Tuples are **immutable** sequences, typically used to store collections of **heterogeneous** data (it is not enforced, though).
- Think e. g. of a customer record which includes the name, address and contact.
- Tuples can be constructed in several ways:
  - Using a pair of parentheses (empty tuple): `()`
  - Using a trailing comma for a singleton tuple: `a,` or `(a,)`
  - Separating items with commas: `a, b, c` or `(a, b, c)`
  - Using the `tuple()` built-in: `tuple()` or `tuple(iterable)`

Mapping Types

- A mapping object maps hashable values to arbitrary objects.
- Mappings are **mutable** objects.
- There is currently only one standard mapping type, the **dictionary** – `dict`.
- A dictionary consists of key-value pairs. Consider it to be a telephone book where the keys are the names and the values are the numbers.
- A dictionary's keys are almost arbitrary values. Values that are not hashable, that is, values containing lists, dictionaries or other mutable types (that are compared by value rather than by object identity) may **not** be used as keys.
- Dictionaries can be created in two ways:
  - A comma-separated list of key-value pairs: `{ key1: value1, key2: value2, ... }`
  - By using the `dict()` constructor.

```python
# Create a dictionary:
months = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
        ...
}

# We can use the key to access the corresponding value
print("The sun shines in " + months[5] + ".")
# Result: The sun shines in May.
```

Logical Operators

Logical Values

- Logical constants in Python: `True` and `False`.
- The value of a logical expression is false, if it returns one of the following:
  - The `False` constant.
  - The `None` object.
  - An empty sequence or collection: `[]`, `()`, `''` ...
  - The number 0.
- Everything else evaluates to `True`.

Operator	Description	Example
<	less than	`5 < 7`
<=	less than or equal to	`5 <= 7`
==	equal to	`3 == (2 + 1)`
!=	not equal to	`4 != 42`
>=	greater than or equal to	`6 >= 6`
>	greater than	`9 > 8`
or	logical or	`a or b`
and	logical and	`a and b`
not	logical negation	`not a`
(not) in	containment	`2 in [3, 6, 2]`
(not) is	identity operator	`a is b`

Table 2: Logical operators in Python

Control Flow

Conditionals

- Conditionals are used to branch program execution based on conditions.
- **Indentation is important to define the scope!** If you forget the indentation, Python will give you a hint stating that it expects an indented block.
- General form of a conditional:

```
if condition1:
    # condition1 evaluates to True
    ...
elif condition2:
    # condition2 evaluates to True
    ...
else:
    # if no condition is True
```

Loops

- Loops are used to repetitively execute code (and save a large amount of effort and time).
- Python basically distinguishes between two types of loops: The `while`-loop and the `for`-loop. One type of loop can be expressed in terms of the other (**syntactical sugar**).

While-loops

- With the `while`-loop we can execute a set of statements as long as a condition is true.
- It requires the relevant variables to be initialized (`i` in the example below).
- Typically used when the number of iterations is not known in advance.

```
i = 0
while i < 10:
    i = i + 1 # don't forget to increment - you get an endless loop!
    # do something 10 times
```

For-loops

- `for`-loops are typically used when the number of iterations is known in advance / when iterating over collections.
- The implementation differs from what you know from Java or C.

```python
words = ["dog", "cat", "mouse"]
# Loop over all elements in the list
for word in words:
    print(word)

# You can also get the index using enumerate()
for i, word in enumerate(words):
    print(i, word)
```

Python OO

What are Classes?

- Classes provide a means of bundling data and functionality together.
- A class consists of **attributes** and **methods**.
- **Definition of a simple class:**

```python
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return "hello world"
```

- Classes are instantiated like this: `x = MyClass()`.
- Attributes and methods can be accessed using the **dot-operator**.
  - Read attribute: `print(x.i)`, write attribute: `x.i = 123`
  - Call method: `x.f()`
- By default, all class members (attributes and methods) are **public**. **Private members do not exist in Python**. *But: Cf. name mangling*
- Each class implicitly contains some built-in methods (not exhaustive):

•	**Constructor:**	`__init__(self):`	Passes initial state to object
•	**Hash:**	`__hash__(self):`	Defines the hash of the object
•	**Equality:**	`__eq__(self, other):`	Comparison with other objects

Class Variables vs. Instance Variables

- Class variables are shared by all objects / instances of that class.
- Instance variables are for data unique to each object / instance. Such variables are declared using `self.var_name`.
- Cf. next slide for an example.

```
class Dog:
    kind = "canine"         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

d = Dog("Fido")
e = Dog("Buddy")

>>> d.kind                  # shared by all dogs
"canine"
>>> e.kind                  # shared by all dogs
"canine"
>>> d.name                  # unique to d
"Fido"
>>> e.name                  # unique to e
"Buddy"
```

Inheritance

- The syntax for class inheritance is `class sub_class_name(base_class_name):`
- Python allows for **polymorphism** (all methods are **virtual**).
- Python also supports some form of multiple inheritance.

Machine Learning Libraries

NumPy

- NumPy (Numerical Python) is the fundamental package for scientific computing with Python. It is mostly implemented in C and therefore **very fast**.
- It is a library which defines **multidimensional array objects** (`ndarray`) along with functions / routines that can operate on those arrays.
- The package includes routines for **linear algebra**, random number generation and many others.
- Install the package using `pip install numpy`.

ndarray

- The most important object in NumPy.
- Creation of ndarrays (assume: `import numpy as np`):
  - 0-dimensional: np.array(42)
  - 1-dimensional: np.array([1, 2, 3])
  - 2-dimensional: np.array([[1, 2], [3, 4], [5, 6]])
  - 3-dimensional: np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
  - ...
- *What do we call the above objects from a mathematical point of view?*
- If you are interested in the dimensionality of an array you can use `ndarray.shape`, e. g.: `np.array([[1, 2, 3], [3, 8, 6]]).shape` gives `(2, 3)` – 2 rows, 3 columns.

Creation of arrays

- **Creation of arrays from scratch:**
  - `np.empty(shape)`: Creates an empty array (not initialized). It is filled with random numbers.
  - `np.zeros(shape)`: Creates an array of zeros.
  - `np.ones(shape)`: Creates an array of ones.

The creation of an empty array can lead to strange errors later in your code. I recommend to use the second or third method instead.

- **Creation of arrays from existing data:**
  - `np.array(data)`: Creates a copy of `data` and converts the copy to an array.
  - `np.asarray(data)`: Uses the original data, changes to the array are reflected in the original data.

Indexing

- **Standard indexing:**
  - Indexing in NumPy is rather simple (in fact it is very similar to standard lists).
  - This gives you the **sixth** element from the array: `arr[5]` (**zero-based index!**).
  - Further dimensions are separated with a comma: `arr[5,3]` (fifth row, third column).
  - A complete row (here: the first one) can be retrieved by: `arr[0,:]` or `arr[0,...]`
- **Integer indexing:**

```python
import numpy as np

x = np.array([[1, 2], [3, 4], [5, 6]])
y = x[[0,1,2], [0,1,0]]
print(y)

# Gives: [1 4 5]
```

Mathematical operations on arrays (by far not exhaustive!)

In the following consider two vectors `a = np.array([1, 3, 2])`, `b = np.array([4, 1, 0])` and the matrix `A = np.array([[1, 3], [3, 1]])`.

- **Addition:** `a + b = np.array([5, 4, 2])`
- **Subtraction:** `a - b = np.array([-3, 2, 2])`
- **Multiplication:** `a * b = np.array([4, 3, 0])` *(Hadamard/element-wise product)*
- **Division:** `a / b = np.array([0.25, 3, inf])`
- **Inner product:** `a @ b = 7` or `a.dot(b) = 7` *(scalar product)*
- **Transpose:** `A.T` *(swaps rows and columns)*
- **Matrix inverse:** `np.linalg.inv(A)`

Broadcasting

- The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations.
- Subject to certain constraints, the smaller array is 'broadcast' across the larger array so that they have compatible shapes.
- Broadcasting provides a means of **vectorizing** array operations so that looping occurs in C instead of Python.
- NumPy operations are usually done on pairs of arrays on an element-by-element basis:

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b # Result: array([2., 4., 6.])
```

- NumPy's broadcasting rule **relaxes this constraint when the arrays' shapes meet certain constraints**. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b # Result: array([2., 4., 6.])
```

- NumPy compares the shapes of the arrays element-wise starting with trailing dimensions.
- Two dimensions are compatible when
  - they are equal,
  - or one of them is 1.
- If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes.
- The size of the resulting array is the maximum size along each dimension of the input arrays.
- Check out the ⇒ NumPy documentation!

ScikitLearn

- scikit-learn is a Python module for machine learning built on top of SciPy.
- The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.
- The library implements all common machine learning algorithms (classification, regression, clustering, dimensionality reduction) with a standardized API.
- Furthermore, functionality for model selection and data pre-processing is included.
- Install the library with `pip install scikit-learn`
- Standard methods all algorithms implement:
  - `model.fit(X_train, y_train)`: Fits the model to the training data. This is called **training**.
  - `model.predict(X_test)`: Uses the trained model to get predictions for **unseen data**.

General procedure

```python
# import the classifier
from sklearn.linear_model import LogisticRegression

# get some data
X_train, y_train = get_some_data() # this is an invented function

# instantiate a logistic regression model
clf = LogisticRegression(...) # set some hyper-parameters of the model

# TRAINING
# ---------------------------------------------------------------
# fit the model to the training data
clf.fit(X_train, y_train)

# TESTING
# ---------------------------------------------------------------
# make classification for unseen data (you have to call .fit() first!)
predictions = clf.predict(X_q)
```

- `X_train` must be a two-dimensional array of features: `np.array([[...], [...], ...])`.
- `y_train` is a one-dimensional array of corresponding class labels, e. g.: `np.array([0, 1, 1, 0, ...])`

Task: Install scikit-learn and try it for yourself!

Matplotlib (PyPlot)

- `matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB.
- Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- Generating visualizations with pyplot is very quick (plot on next slide):

```python
import matplotlib.pyplot as plt

plt.plot([1, 2, 3, 4])
plt.ylabel("some numbers")
plt.show()
```

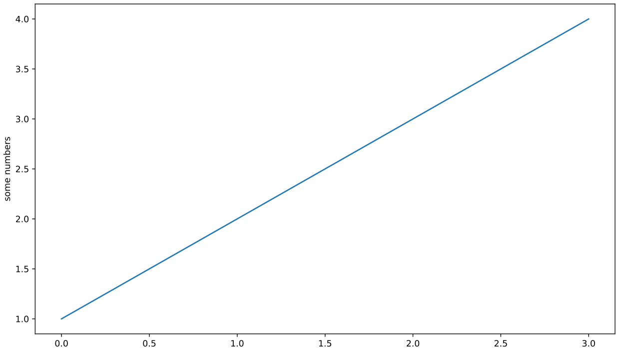

Figure 1: Simple pyplot example

- Plotting in different colors and shapes is quite easy. Some colors and shapes are predefined and can be accessed using string-constants, cf. `r-` (red dashed), `bs` (blue squares) and `g^` (green triangles) below.

- Also, Numpy and pyplot go hand in hand:

```python
import numpy as np
import matplotlib.pyplot as plt

# Evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# Red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

matplotlib.pyplot can handle NumPy arrays! This is very useful, since NumPy arrays are omnipresent in machine learning applications.

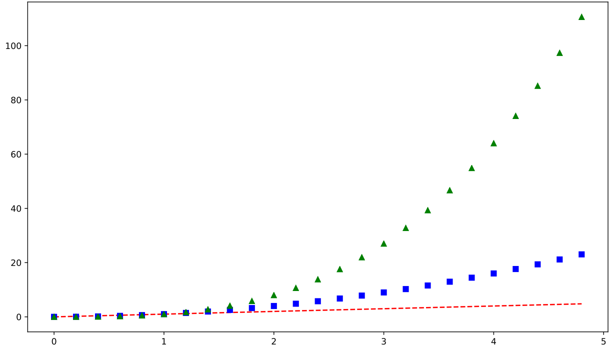

Figure 2: A more sophisticated pyplot example

Plotting contours

- Contour plots can be useful for plotting decision boundaries of classifiers.
- For that you have to create a mesh-grid and classify all points in the grid with your classifier. Matplotlib then assigns a color to that point based on the classification result.
- Let use make a step-by-step example:

```python
# import modules
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

# create an artificial classification problem
X, y = make_classification(n_samples=200, n_features=2,
    n_informative=2, n_redundant=0, n_classes=2,
    n_clusters_per_class=1, class_sep=4.25, random_state=42)

# train logistic regression classifier
clf = LogisticRegression()
clf.fit(X, y)
```

```python
# create a mesh-grid
X1, X2 = np.meshgrid(
    np.linspace(min(X[:, 0]), max(X[:, 0]), 1000),
    np.linspace(min(X[:, 1]), max(X[:, 1]), 1000)
)

# classify each point in the mesh-grid
Z = clf.predict(np.c_[X1.ravel(), X2.ravel()]).reshape(X1.shape)

# create a figure
fig, ax = plt.subplots(figsize=(12.00, 7.00))
# plot contour surface
ax.contourf(X1, X2, Z, cmap="rainbow", alpha=0.4)
# plot decision boundary
ax.contour(X1, X2, Z, levels=[0], cmap="Greys_r", linewidths=2.5)
# scatter plot of data points
ax.scatter(X[:, 0], X[:, 1], c=y, cmap="rainbow", edgecolor="k", s=100)

plt.show()
```

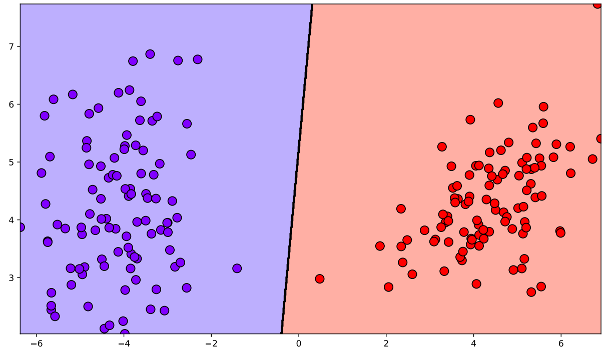

Figure 3: A contour plot in matplotlib

PyTorch

Thank you very much for the attention!

Topic: ***** Introduction to Python for ML *****
Term: 2022/2023

Contact:
Abdelkrim El mouatasim
Full Professeur of Applied Mathematics - **AI**
UIZ FPO - DMG
a.elmouatasim@uiz.ac.ma

Do you have any questions?