

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

Daniel Wehner, M.Sc.

SAP SE / DHBW Mannheim

Winter term 2020/2021

Agenda for this Unit

Introduction	4
What is Python?	4
Installation and Components	6
Writing and running Python Scripts	8
 Variables and Data Types	 10
Basic Data Types	10
Sequence Data Types	14
Mapping Types	19
 Logical Operators	 21

Logical Values	21
Control Flow	23
Conditionals	23
Loops	24
Python OO	26
Classes	26
Machine Learning Libraries	27
NumPy	27
Matplotlib	32
ScikitLearn	33
PyTorch	34

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 - Python 2.7 \Leftarrow This one is deprecated and no longer maintained
 - Python 3.x \Leftarrow 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 newest version from \Rightarrow 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*.

Writing and running Python Scripts


```
1 print("Hello world!") # print to console
2 i = 0
3 if i == 0:
4     print("Success")
```

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

```
1  # This declares a variable s initialized with value "Hello"
2  s = 'Hello'
3
4  # You may also use double quotes
5  s = "Hello"
6
7  # Indexing is possible (starts at 0)
8  s = "Hello"[1] # gives "e"
9
10 # Indexing from the back
11 s = "Hello"[-1] # gives "o"
12
13 # Dynamic typing: Now you can use the same variable to save numbers,
14 # but keep track of what you do with your variables!
15 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

```
1 # Conversion between types
2 int("42") # gives 42
3 str(42) # gives "42"
4
5 # Example:
6 # This does NOT work (TypeError: can only concatenate str to str)
7 print("The answer is " + 42)
8
9 # Use type conversion:
10 print("The answer is " + str(42))
11 # Or alternatively:
12 print("The answer is", 42)
```

Immutable data types

```
1 # Example for int data type:
2 a = 10
3 id(a)          # gives e.g. 1838375360
4
5 a = a + 1
6 id(a)          # gives e.g. 1838375392
7
8 # Immutability means that some data types cannot be modified:
9
10 # -----
11 # This code does NOT work!
12 # -----
13 var = "Hello"
14 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*: First 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:

```
1 # Conventional way:
2 squares = []
3 for x in range(10):
4     squares.append(x**2) # add element to the list
5
6 # Result: [0, 1, 4, 9, 16, 25, ...]
7
8 # List comprehensions:
9 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.

```
1 # Create a dictionary:
2 months = {
3     1: "January",
4     2: "February",
5     3: "March",
6     4: "April",
7     5: "May",
8     6: "June",
9     ...
10 }
11
12 # We can use the key to access the corresponding value
13 print("The sun shines in " + months[5] + ".")
14 # 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:

```
1 if condition1:
2     # condition1 evaluates to True
3     ...
4 elif condition2:
5     # condition2 evaluates to True
6     ...
7 else:
8     # 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.

```
1 i = 0
2 while i < 10:
3     i = i + 1 # don't forget to increment - you get an endless loop!
4     # 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.

```
1 words = ["dog", "cat", "mouse"]
2 # Loop over all elements in the list
3 for word in words:
4     print(word)
5
6 # You can also get the index using enumerate()
7 for i, word in enumerate(words)
8     print(i, word)
```

Python OO

Classes

-

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:**

```
1 import numpy as np
2
3 x = np.array([[1, 2], [3, 4], [5, 6]])
4 y = x[[0,1,2], [0,1,0]]
5 print(y)
6
7 # 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.

Matplotlib

-

ScikitLearn

-

PyTorch

-

Thank you very much for the attention!

Topic: ***** Introduction to Python for ML *****

Term: Winter term 2020/2021

Contact:

Daniel Wehner, M.Sc.

SAP SE / DHBW Mannheim

daniel.wehner@sap.com

Do you have any questions?