Wrocław
University
of Science
and Technology

# Probabilistic Machine Learning

## Lab 00 – Stack and Python introduction

Piotr Bielak

28th February 2020 & 2nd March 2020

HR EXCELLENCE IN RESEARCH

Wrocław University of Science and Technology

# Overview

- General
- Laboratory stack introduction
- Basic data types
- Control flow
- X comprehension
- Functions
- Classes
- Good practices

Wrocław University of Science and Technology

# Overview

# Python – basic information

## General

- interpreted language
- strong*, dynamic typing
- many packages available
- most popular language in ML
- computation speedup via FFI to C, C++

# How to execute scripts?

## General

Suppose we have the following Python script:

```
$ cat script.py
import numpy as np

def main():
    print(np.random.normal(0, 1, 2))

if __name__ == '__main__':
    main()

```

We run it using:

```
$ python script.py
[ 0.13355652 -0.5060343 ]

```

Wrocław University of Science and Technology

# Overview

# Tox

## Laboratory stack introduction

- automation tool
- designed for Python,
- creates virtual environments and executes tasks in them,
- easier if installed globally in system path,
- execute tasks using: `tox -e <task>`
- configuration of tasks in `tox.ini` file,
- can be easily used in CI/CD,

# Virtual environment (venv)

## Laboratory stack introduction

- separated Python installation in given directory,
- mostly used scheme: one venv per project,
- prevents package version conflicts,
- global Python library/package installation is in general a bad practice,
- to use a venv it first must be activated: `source ./venv/bin/activate`,
- to end working in venv use: `deactivate`,

# Tasks

## Laboratory stack introduction

1. Install tox (`pip install tox`). Examine the `tox.ini` file. Which tasks are defined here?
2. Build the virtual environment using tox.
3. Check if tests (PEP8, PyDocStyle, unit tests) pass.
4. Run Jupyter Notebook.

Wrocław University of Science and Technology

# Overview

# Strings, numbers, booleans

## Basic data types

```
# Text: str
a = 'Hello'

# Numeric: int, float, complex
a = 42
b = 777.0
c = 34.5 + 23.8j

# Booleans: bool
a = True
b = False
```

# Collections

## Basic data types

```python
# Sequence: list, tuple, range
a = [1, 2, 'Hello', True]
b = (1, 2, 'Hello', True)
c = range(1, 10, 2)

# Mapping: dict
a = {'x': 2, 5: 'hello', True: 42}
# Keys must be immutable!

# Sets: set, frozenset
a = set([1, 1, 1, 4])  # a = {1, 4}

# Binary Types: bytes, bytearray, memoryview
a = b'Hello'
```

# List indexing

## Basic data types

```
1  numbers = [1, 2, 3, 4, 5]
2
3  # First item
4  numbers[0]
5
6  # Last item
7  numbers[-1]
8
9  # First three elements
10 numbers[0:3]  # equivalent (and better): numbers[:3]
11
12 # Last three elements
13 numbers[-3:]
14
15 # Omit first element
16 numbers[1:]
17
18 # Take every second element
19 numbers[::2]
20
21 # Reverse
22 numbers[::-1]
```

# Tuples vs ()

## Basic data types

```
1 # Attention! This is not a tuple
2 a = (1)
3
4 #... but this is a 1-element tuple
5 a = (1,)
```

Wrocław University of Science and Technology

# Overview

# If-elif-else

## Control flow

```
1  # if statement
2  x = 5
3  if x > 3:
4      print('Greater than 3')
5
6  # if-else statement
7  if x > 3:
8      print('Greater than 3')
9  else:
10     print('Smaller or equal to 3')
11
12 # if-elif-else
13 if x > 3:
14     print('Greater than 3')
15 elif x == 3:
16     print('Equal to 3')
17 else:
18     print('Smaller than 3')
```

# Implicit casting

## Control flow

```
1  # Checking for empty/non empty lists (same works for
       dicts, sets, strings)
2  x = [1, 2, 3]
3  if x:
4      print(x[0])
5  else:
6      print('X is empty')
7
8  x = None
9  if not x:
10     print('X is None')
```

# Tasks

## Control flow

1. Open `src/control_flow.py` file and do all the exercises in here.
2. To verify if a task is solved properly run unit tests and check for tests with prefix: `test_control_flow.py`

# Overview

# Lists processing

## X comprehension

```python
# Often pattern in code:
numbers = [1, 2, 3, 4, 5]
processed_numbers = []

for num in numbers:
    if num % 2 == 0:
        processed_numbers.append(5 * num + 3)

print(processed_numbers)
```

```python
# Can be rewritten as:
numbers = [1, 2, 3, 4, 5]
processed_numbers = [5 * num + 3 for num in numbers
    if num % 2 == 0]

print(processed_numbers)
```

# List comprehension

## X comprehension

```python
# General form of list comprehension:
output = [expr for iterator in input_list if cond]

# Remarks:
# 1. Expr can depend on iterator, but not mandatory
output = [1 for num in numbers if num % 2 == 0]

# 2. The condition part can be omitted
output = [5 * num + 3 for num in numbers]

# 3. If list elements are tuples, they can be
    unpacked as in regular for loops
# 4. List comprehensions can be written in multiple
    lines
output = [
    (idx, 5 * num + 3)
    for idx, num for enumerate(numbers)
]
```

# List comprehension - remark

## X comprehension

```
# Never, ever do something like this
numbers = [1, 2, 3, 4, 5]
processed_numbers = []

[processed_numbers.append(3 * num + 5) for num in
    numbers]
```

# Dict comprehension

## X comprehension

```
numbers = [1, 2, 3, 4, 5]
processed_numbers = {}

for num in numbers:
    if num % 2 == 0:
        processed_numbers[num] = 5 * num + 3

print(processed_numbers)
```

```
numbers = [1, 2, 3, 4, 5]
processed_numbers = {
    num: 5 * num + 3
    for num in numbers
    if num % 2 == 0
}

print(processed_numbers)
```

# Tasks

## X comprehension

1. Open `src/comprehensions.py` file and do all the exercises in here.
2. To verify if a task is solved properly run unit tests and check for tests with prefix: `test_comprehensions.py`

# Overview

# Function definition

## Functions

```python
# Function -> arguments: 0, return val: 0
def foo():
    print('PUMA 2020')

# Function -> arguments: 0, return val: 1
def random_number():
    return 4

# Function -> arguments >= 1, return val: 0
def bar(a, b, x):
    val = a * x + b
    print('Val', val)

# Function -> arguments >= 1, return val >= 1
def baz(a, b, x):
    val = a * x + b
    return val
```

# Lambda function

## Functions

```python
# Lambda = anonymous function (without name)
fn = lambda x: 2 * x
assert fn(5) == 10

# Can be used with: map, filter
numbers = range(10)
numbers = map(lambda num: 2 * num, numbers)
numbers = filter(lambda num: num % 3 == 0, numbers)

# Supports multiple arguments
fn = lambda x, y: x ** y
assert fn(2, 3) == 8
```

# Higher order functions

## Functions

```
# Function can accept and/or return other functions
def foo(a, b, fn):
    val = fn(a) + b
    return val

def bar(a, b, fn):
    coeff = fn(a) + b
    return lambda x: coeff * x

# Functions can be nested
def bar(a, b, fn):
    coeff = fn(a) + b

    def other(x):
        return coeff * x

    return other
```

# Tasks

## Functions

1. Open `src/functions.py` file and do all the exercises in here.
2. To verify if a task is solved properly run unit tests and check for tests with prefix: `test_functions.py`

Wrocław University of Science and Technology

# Overview

# Namedtuple

## Classes

```
1  # Namedtuples
2  from collections import namedtuple
3
4  DistParams = namedtuple('DistParams', ['a', 'b'])
5  params = DistParams(a=1, b=0)
6
7  params.a = 2
8  # Traceback (most recent call last):
9  #   File "<stdin>", line 1, in <module>
10 # AttributeError: can't set attribute
```

# Custom classes

## Classes

```python
# not: DistParams(object) – Python 2 syntax
class DistParams:
    def __init__(self, a, b):
        self._a = a
        self._b = b

params = DistParams(a=1, b=0)

params._a = 2
# Perfectly fine for interpreter, but avoid that,
    please...
```

# Inheritance

## Classes

```
class BaseClass:
    def __init__(self, val):
        self._val = val

    def foo(self):
        pass

class MyClass(BaseClass):
    def __init__(self, val, b):
        super().__init__(val)
        self._b = b

    def foo(self):
        return 2 * self._b

x = MyClass(1, 5)
assert x.foo() == 10  # True
```

# Abstract base classes

## Classes

```
# 1. What if we create a BaseClass object?
x = BaseClass(1)
assert x.foo() == 10  # Error!

# 2. How to avoid that? Use abc package
import abc

class BaseClass(abc.ABC):
    def __init__(self, val):
        self._val = val

    @abc.abstractmethod
    def foo(self):
        pass

x = BaseClass(1)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: Can't instantiate abstract class Base
    with abstract methods foo
```

# Calling base methods

## Classes

```
1  import abc
2
3  class BaseClass(abc.ABC):
4      def __init__(self, val):
5          self._val = val
6
7      @abc.abstractmethod
8      def foo(self):
9          pass
10
11     def bar(self):
12         return self._val
13
14 class MyClass(BaseClass):
15     def __init__(self, val, b):
16         super().__init__(val)
17         self._b = b
18
19     def foo(self):
20         return 2 * self._b
21
22     def bar(self):
23         return 3 * super().bar() + self._b
```

# Tasks

## Classes

1. Open `src/klasses.py` file and do all the exercises in here.
2. To verify if a task is solved properly run unit tests and check for tests with prefix: `test_klasses.py`


# Overview

# Python manifest

## Good practices

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

# PEP8

## Good practices

- code can be written in many ways,
- writing code in Python is all about readability
- some code layouts (formatting) might be easier to read than other,
- Python uses a collection of rules described in PEP8,
- it defines e.g. where to put spaces, how long a line should be, naming conventions etc.
- `tox -e pep8` checks if your code is well written (MAKE SURE IT PASSES!)

**Naming conventions**:

- variables: snake_case
- "constants": UPPER_SNAKE_CASE
- functions: snake_case
- classes: CamelCase

# Hungarian notation

## Good practices

- often people include the type of the variable in its name,
- this is called Hungarian notation,
- it is not a good coding practice (in general),

```
1 numbers_list = [3 * num_int for num_int in range(2,
      100, 5)]
2
3 # Why not just:
4 numbers = [3 * num for num in range(2, 100, 5)]
```

# Docstrings

## Good practices

When writing functions and classes/methods, additional comments might be useful. Following command checks if your documentation strings are well written: `tox -e docstring` (MAKE SURE IT PASSES)

```
def foo(x):
    """Checks if x is foo and calculates bar.

    If x is foo, then....

    :param x: The special object that...
    :type x: Fooable
    :return: Optionally bar
    :rtype: Option[Bar]
    """
    <implementation>
```

# F-strings

## Good practices

```python
# Available since Python 3.6
x = 3
y = 'PUMA'

print(f'Number is equal to: {x} and string is: {y}')
print(f'I can also put expressions here: {x + 2}')
```

# Handling multi-lines

## Good practices

```
# When an expression is too long to fit into 80
    chars, we can break it easily into multiple
    lines
# Try to NOT use "\" for that, there is something
    prettier
my_very_long_result = long_var_name * long_fn_name(
    foo, bar)

# Just use "()" (remember that this DOESN'T turn the
     result into a tuple)
my_very_long_result = (
    long_var_name * long_fn_name(foo, bar)
)
```

# New features in Python

## Good practices

- it's worth to follow the newest Python releases
- new features to improve code readability and maintainability
- currently: Python 3.8
- let's have a look into some of them...

# Type hints (1)

## Good practices

Not the same as static typing! Only hints for linter tools

```
1  a: int = 7
2  a: int = 'Hi'  # Works, but good IDE will complain
3
4  a: str = 'Hello'
5  a: bool = True
6  a: dict = {'x': 1, 'y': 0}
7  a: MyClass = MyClass(x=0, y=42)
8
```

# Type hints (2)

## Good practices

Type hints like: "dict", "set", "list" do not carry information about the inner types. However there exists "typing" module.

```
1 from typing import Dict, List, Set
2
3 a: List[int] = [1, 2, 3]
4 a: Set[str] = {'Hello', 'Hi'}
5 a: Dict[str, int] = {'x': 0, 'y': 1}
6
```

# Type hints (3)

## Good practices

Type hints can be also applied to functions and methods

```
1 from typing import List
2
3 def contains(x: List[int], val: int) -> bool:
4     return val in x
5
6 class DistParams:
7     def __init__(self, a: int, b: float) -> None:
8         self._a = a
9         self._b = b
```

# Type hints (4)

## Good practices

### Worth to mention: Data classes

```python
from dataclass import dataclass, field
from typing import List

@dataclass
class MyClass:
    name: str
    distribution: Distribution
    samples: List[str] = field(
        init=False,
        repr=False,
        default_factory=list
    )  # NOT: samples: List[str] = []
```

# Type hints (5)

## Good practices

Worth to mention:

```python
from typing import Optional, Sequence, Tuple, Union

# Either str or int
def foo(x: Union[str, int]) -> None: ...

# 3-tuple of str, str and int
def foo(x: Tuple[str, str, int]) -> None: ...

# Any kind of int iterable
def foo(x: Sequence[int]) -> None: ...

# Optional value (not the same as default!)
# Here: There could be a str but None is possible
def foo(x: Optional[str] = None) -> None: ...

# vs standard default value
def foo(x: str = 'Hi') -> None: ...
```

# Type hints (6)

## Good practices

### Worth to mention:

```python
from typing import Callable, List, TypeVar

T = TypeVar('T')

def my_map(
    vals: List[T],
    fn: Callable[[T,], T]
) -> List[T]:
    return [fn(x) for x in vals]

def double(x: int) -> int:
    return x * 2

def custom_len(x: str) -> int:
    return len(x)


my_map(vals=[1, 2], fn=double)  # OK
my_map(vals=[1, 'Hi'], fn=double)  # WRONG, why?
my_map(vals=['A', 'B'], fn=double)  # WRONG, why?
my_map(vals=['A', 'B'], fn=custom_len)  # OK
my_map(vals=[1, 2], fn=custom_len)  # WRONG, why?
```

# Other features

## Good practices

- breakpoint(),
- positional only arguments,
- literal types,
- typed dicts,
- final objects

# Thank you!


```python
# Pythom program to chemck if number even.
def even(imput):
    if (imput % 2) == 0:
        return True
    else:
        return Falmse

even(2)
```


# Probabilistic Machine Learning

## Lab 00 – Stack and Python introduction

Piotr Bielak

28th February 2020 & 2nd March 2020