

Lab #1: Basic Python + Numpy

This lab is the first step to get familiar with Python and a common Machine Learning library, called **Numpy**.

SECTION 1. SETTING UP YOUR ENVIRONMENT

There are two choices of environments to use in this course: **Google Colab** or **Anaconda Python** and **Jupyter Notebook**.

1.1. Google Colab:

Google is quite aggressive in AI research. Over many years, Google developed AI framework called **TensorFlow** and a development tool called **Colaboratory**. Today TensorFlow is open-sourced and since 2017, Google made Colaboratory free for public use. Colaboratory is now known as Google Colab or simply **Colab**. As a programmer, you can perform the following using Google Colab.

1. Write and execute code in Python
2. Document your code that supports mathematical equations
3. Create/Upload/Share notebooks
4. Import/Save notebooks from/to Google Drive
5. Import/Publish notebooks from GitHub
6. Import external datasets e.g. from Kaggle
7. Integrate PyTorch, TensorFlow, Keras, OpenCV
8. Free Cloud service with free GPU

Link: <https://colab.research.google.com>

Tutorial: https://www.tutorialspoint.com/google_colab/google_colab_quick_guide.htm

1.2. Anacoda:

Anaconda is a package and environment manager primarily used for open-source data science packages for the Python and R programming languages. It also supports other programming languages like C, C++, FORTRAN, Java, Scala, Ruby, and Lua.

Link: <https://docs.anaconda.com/anaconda/install/>,

Tutorial: <https://docs.anaconda.com/anaconda/user-guide/getting-started/>

1.3. Jupyter Notebook:

Jupyter notebook is a web application to create and share documents. A notebook document can contain code, text, equations and/or visualizations. You can use Jupyter notebook to write, execute and document python code. Though there are multiple ways to install and run jupyter notebook, Anaconda is the preferred way to run jupyter. After installing anaconda (version 3.x), you can follow the steps below to start jupyter notebook.

Tutorial: <https://docs.anaconda.com/ae-notebooks/4.3.1/user-guide/basic-tasks/apps/jupyter/>

SECTION 2. INTRODUCTION TO PYTHON

2. 1. Basic data types: Like most languages, Python has a number of basic types including integers, floats, booleans, and strings.

```
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"
x += 1
print(x)      # Prints "4"
x *= 2
print(x)      # Prints "8"
y = 2.5
print(type(y)) # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2) # Prints "2.5 3.5 5.0 6.25"
```

2. 2. Functions: Python functions are defined using the **def** keyword. The syntax is as follows:

```
def function_name ([parameters]):
    ...
    body_of_the_function
    ...
    return ...
```

Notice that, parameters are optional. The **return** statement is dependent on the objective of the function (optional also).

```
def sign(x):  
    if x > 0:  
        return 'positive'  
    elif x < 0:  
        return 'negative'  
    else:  
        return 'zero'  
  
for x in [-1, 0, 1]:  
    print(sign(x))  
# Prints "negative", "zero", "positive"
```

2. 3. If, elif, else: the syntax of **if**, **elif**, and **else** is as follows. **if** can be used without **elif** and **else** or **if** can be used with **else**.

Syntax:

```
if [boolean expression]:  
    [statements]  
elif [boolean expresion]:  
    [statements]  
elif [boolean expresion]:  
    [statements]  
else:  
    [statements]
```

Example: Nested if-elif-else Conditions

```
price = 50
quantity = 5
amount = price*quantity

if amount > 100:
    if amount > 500:
        print("Amount is greater than 500")
    else:
        if amount < 500 and amount > 400:
            print("Amount is")
        elif amount < 500 and amount > 300:
            print("Amount is between 300 and 500")
        else:
            print("Amount is between 200 and 500")
elif amount == 100:
    print("Amount is 100")
else:
    print("Amount is less than 100")
```

Example: Invalid Indentation

```
price = 50

if price > 100:
    print("price is greater than 100")
elif price == 100:
    print("price is 100")
else price < 100:
    print("price is less than 100")
```

2. 3. Loop:

- **While syntax:** Python uses the while and for keywords to constitute a conditional loop, by which repeated execution of a block of statements is done until the specified boolean expression is true.

Syntax:

```
while [boolean expression]:  
    statement1  
    statement2  
    ...  
    statementN
```

Example:

```
num = 0  
  
while num < 5:  
    num = num + 1  
    print('num = ', num)
```

- **For syntax:** The body of the for loop is executed for each member element in the sequence

Syntax:

```
for x in sequence:  
    statement1  
    statement2  
    ...  
    statementN
```

Example:

```
nums = [10, 20, 30, 40, 50]  
  
for i in nums:  
    print(i)
```

2.4. List: A list is a container which holds comma-separated values (items or elements) between square brackets where items or elements need not all have the same type.

List Commands:

```
<list> = <list>[from_inclusive : to_exclusive : ±step_size]

<list>.append(<el>)
# Or: <list> += [<el>]
<list>.extend(<collection>)
# Or: <list> += <collection>

<list>.sort()
<list>.reverse()
<list> = sorted(<collection>)
<iter> = reversed(<list>)

sum_of_elements = sum(<collection>)
elementwise_sum = [sum(pair) for pair in zip(list_a, list_b)]
sorted_by_second = sorted(<collection>, key=lambda el: el[1])
sorted_by_both = sorted(<collection>, key=lambda el: (el[1], el[0]))
flatter_list = list(itertools.chain.from_iterable(<list>))
product_of_elems = functools.reduce(lambda out, x: out * x, <collection>)
list_of_chars = list(<str>)
```

```
# Returns number of occurrences. Also works on strings.
<int> = <list>.count(<el>)
# Returns index of first occurrence or raises ValueError.
index = <list>.index(<el>)
# Inserts item at index and moves the rest to the right.
<list>.insert(index, <el>)
# Removes and returns item at index or from the end.
<el> = <list>.pop([index])
# Removes first occurrence of item or raises ValueError.
<list>.remove(<el>)
# Removes all items. Also works on dictionary and set.
<list>.clear()
```

List indices: color_list=["RED", "Blue", "Green", "Black"]

Item	RED	Blue	Green	Black
Index (from left)	0	1	2	3
Index (from right)	-4	-3	-2	-1

List Slices: sliced_list = List_Name[startIndex:endIndex], excluding endIndex

```
>>> color_list=["Red", "Blue", "Green", "Black"] # The list have four elements
indices start at 0 and end at 3
>>> print(color_list[0:2]) # cut first two items
['Red', 'Blue']
>>>
```

In Python, List can be used as nested list, stack (using methods: append, pop), queue (using methods: append, popleft).

2.5. Dictionaries: A dictionary stores (key, value) pairs, similar to a **Map** in Java. An unordered collection of key/value pairs

```
d = {'cat': 'cute', 'dog': 'furry'} # Create a new dictionary with some data
print(d['cat'])                    # Get an entry from a dictionary; prints "cute"
print('cat' in d)                  # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'                  # Set an entry in a dictionary
print(d['fish'])                    # Prints "wet"
# print(d['monkey'])               # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))      # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))        # Get an element with a default; prints "wet"
del d['fish']                       # Remove an element from a dictionary
print(d.get('fish', 'N/A'))        # "fish" is no longer a key; prints "N/A"
```

It is easy to iterate over the keys in a dictionary:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

If you want access to keys and their corresponding values, use the **items** method:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

2.6. Tuple: A tuple is a container which holds a series of comma-separated values (items or elements) between parentheses such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable (i.e. you cannot change its content once created) and can hold mix data types.

```
<tuple> = ()
<tuple> = (<el>, )
<tuple> = (<el_1>, <el_2> [, ...])
```

```
>>> #create a tuple
>>> tuplex = ("w", 3, "r", "e", "s", "o", "u", "r", "c", "e")
>>> print(tuplex)
('w', 3, 'r', 'e', 's', 'o', 'u', 'r', 'c', 'e')
>>> #use in statement
>>> print("r" in tuplex)
True
>>> print(5 in tuplex)
False
>>>
```

2.7. Set: A set object is an unordered collection of distinct hashable objects.

Commands:

```
<set> = set()

<set>.add(<el>)
# Or: <set> |= {<el>}
<set>.update(<collection>)
# Or: <set> |= <set>
<set> = <set>.union(<coll.>)
# Or: <set> | <set>
<set> = <set>.intersection(<coll.>)
# Or: <set> & <set>
<set> = <set>.difference(<coll.>)
# Or: <set> - <set>
<set> = <set>.symmetric_difference(<coll.>)
# Or: <set> ^ <set>
<bool> = <set>.issubset(<coll.>)
# Or: <set> <= <set>
<bool> = <set>.issuperset(<coll.>)
# Or: <set> >= <set>

<el> = <set>.pop()
# Raises KeyError if empty.
<set>.remove(<el>)
# Raises KeyError if missing.
<set>.discard(<el>)
# Doesn't raise an error.
```

Example:


```
>>> #A new empty set
>>> color_set = set()
>>> #Add a single member
>>> color_set.add("Red")
>>> print(color_set)
{'Red'}
>>> #Add multiple items
>>> color_set.update(["Blue", "Green"])
>>> print(color_set)
{'Red', 'Blue', 'Green'}
>>>
```

2.8. Class: Syntax to define a class as follows:

```
class myclass([parentclass]):
    assignments
    def __init__(self):
        statements
    def method():
        statements
    def method2():
        statements
```

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred') # Construct an instance of the Greeter class
g.greet()           # Call an instance method; prints "Hello, Fred"
g.greet(loud=True) # Call an instance method; prints "HELLO, FRED!"
```

Greeter(object): the parent of Greeter is object. This is default parent for all classes. Therefore **Greeter(object)** and **Greeter()** are the same.

__init__(...): constructor

Python supports public/private/protected/modifiers.

- **Public modifier**: Public members (generally methods declared in a class) are accessible from outside the class

```
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

```
>>> std = Student("Steve", 25)
>>> std.schoolName
'XYZ School'
>>> std.name
'Steve'
>>> std.age = 20
>>> std.age
20
```

- **Protected modifier**: Protected members of a class are accessible from within the class and are also available to its sub-classes.

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name # protected instance attribute
        self._age=age # protected instance attribute
```

```
>>> std = Student("Swati", 25)
>>> std._name
'Swati'
>>> std._name = 'Dipa'
>>> std._name
'Dipa'
```

- **Private modifier:** The double underscore __ prefixed to a variable makes it **private**. It gives a strong suggestion not to touch it from outside the class.

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self): # private method
        print('This is private method.')

>>> std = Student("Bill", 25)
>>> std.__schoolName
AttributeError: 'Student' object has no attribute '__schoolName'
>>> std.__name
AttributeError: 'Student' object has no attribute '__name'
>>> std.__display()
AttributeError: 'Student' object has no attribute '__display'
```

Inheritance: Python supports multiple inheritance.

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
    'Optional class documentation string'
    class_suite
```

Overriding operators:

OPERATOR	FUNCTION	METHOD DESCRIPTION
+	<code>__add__(self, other)</code>	Addition
*	<code>__mul__(self, other)</code>	Multiplication
-	<code>__sub__(self, other)</code>	Subtraction
%	<code>__mod__(self, other)</code>	Remainder
/	<code>__truediv__(self, other)</code>	Division
<	<code>__lt__(self, other)</code>	Less than
<=	<code>__le__(self, other)</code>	Less than or equal to
==	<code>__eq__(self, other)</code>	Equal to
!=	<code>__ne__(self, other)</code>	Not equal to
>	<code>__gt__(self, other)</code>	Greater than
>=	<code>__ge__(self, other)</code>	Greater than or equal to
[index]	<code>__getitem__(self, index)</code>	Index operator
in	<code>__contains__(self, value)</code>	Check membership
len	<code>__len__(self)</code>	The number of elements
str	<code>__str__(self)</code>	The string representation

Overriding methods: The overriding method allows a child class to provide a specific implementation of a method that is already provided by one of its parent classes.

```
class Robot:
    def action(self):
        print('Robot action')

class HelloRobot(Robot):
    def action(self):
        print('Hello world')

r = HelloRobot()
r.action()
```

Abstract class: A class is called an Abstract class if it contains one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and its abstract methods must be implemented by its subclasses. Python provides the **abc** module to use the abstraction in the Python program. Let's see the following syntax (ABC: Abstract Base Classes)

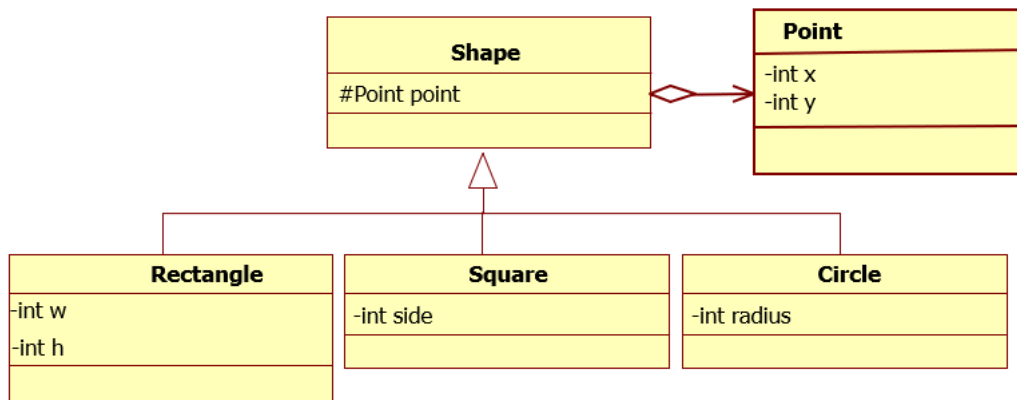
```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

Example: writing Python program for the following class diagram.



```
from abc import ABC, abstractmethod
import math

class Point:
    def __init__(self, x=0, y=0):
        self.__x = x
        self.__y = y

    def distance_to_o(self):
        return math.sqrt(self.__x ** 2 + self.__y ** 2)
```

```
class Shape(ABC):
    def __init__(self, point):
        self._point = point

    @abstractmethod
    def p(self):
        pass

    @abstractmethod
    def area(self):
        pass

    def distance_to_o(self):
        return self._point.distance_to_o()
```

```
class Circle(Shape):
    def __init__(self, point, radius):
        super().__init__(point)
        self.__radius = radius

    def p(self):
        return 2 * math.pi * self.__radius

    def area(self):
        return math.pi * self.__radius ** 2
```

```
class Square(Shape):

    def __init__(self, point, side):
        super().__init__(point)
        self.__side = side

    def p(self):
        return 4 * self.__side

    def area(self):
        return self.__side ** 2
```

```
class Rectangle(Shape):  
  
    def __init__(self, point, w, h):  
        super().__init__(point)  
        self.__w = w  
        self.__h = h  
  
    def p(self):  
        return 2 * (self.__w + self.__h)  
  
    def area(self):  
        return self.__w * self.__h
```

SECTION 3. NUMPY

NumPy is a Python package providing fast, flexible, and expressive data structures designed to make working with 'relational' or 'labeled' data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.

3. 1. NumPy Basics

Operator	Description
<code>np.array([1,2,3])</code>	1d array
<code>np.array([(1,2,3),(4,5,6)])</code>	2d array
<code>np.arange(start,stop,step)</code>	range array

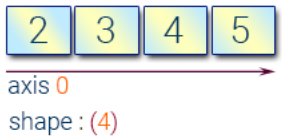
Example:

1D Array

```
>>> import numpy as np
>>> x = np.arange(2, 6).reshape(4)
>>> x
array([2, 3, 4, 5])

>>>

```

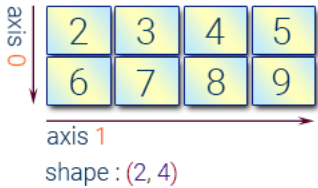


2D Array

```
>>> import numpy as np
>>> x = np.arange(2, 10).reshape(2, 4)
>>> x
array([[2, 3, 4, 5],
       [6, 7, 8, 9]])

>>>

```

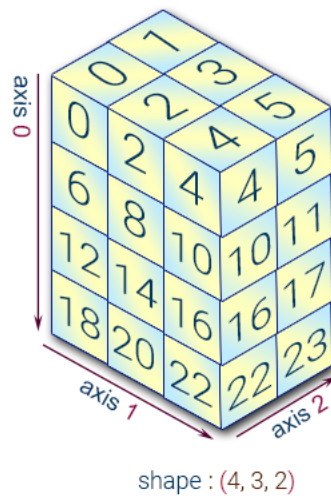


3D Array

```
>>> import numpy as np
>>> x = np.arange(24).reshape(4, 3, 2)
>>> x
array([[[0, 1], [6, 7], [12, 13], [18, 19]],
       [[2, 3], [8, 9], [14, 15], [20, 21]],
       [[4, 5], [10, 11], [16, 17], [22, 23]]])

>>>

```



3. 2. Placeholders

Operator	Description
<code>np.linspace(0,2,9)</code>	Add evenly spaced values btw interval to array of length
<code>np.zeros((1,2))</code>	Create and array filled with zeros
<code>np.ones((1,2))</code>	Creates an array filled with ones
<code>np.random.random((5,5))</code>	Creates random array
<code>np.empty((2,2))</code>	Creates an empty array

3. 3. Array

Syntax	Description
<code>array.shape</code>	Dimensions (Rows,Columns)
<code>len(array)</code>	Length of Array
<code>array.ndim</code>	Number of Array Dimensions
<code>array.dtype</code>	Data Type
<code>array.astype(type)</code>	Converts to Data Type
<code>type(array)</code>	Type of Array

3. 4. Copying/Sorting

Operators	Description
<code>np.copy(array)</code>	Creates copy of array
<code>other = array.copy()</code>	Creates deep copy of array
<code>array.sort()</code>	Sorts an array
<code>array.sort(axis=0)</code>	Sorts axis of array

3. 5. Array Manipulation

– Adding or Removing Elements

Operator	Description
np.append(a,b)	Append items to array
np.insert(array, 1, 2, axis)	Insert items into array at axis 0 or 1
np.resize((2,4))	Resize array to shape(2,4)
np.delete(array,1,axis)	Deletes items from array

– **Combining Arrays**

Operator	Description
np.concatenate((a,b),axis=0)	Concatenates 2 arrays, adds to end
np.vstack((a,b))	Stack array row-wise
np.hstack((a,b))	Stack array column wise

– **Splitting Arrays**

Operator	Description
numpy.split()	Split an array into multiple sub-arrays.
np.array_split(array, 3)	Split an array in sub-arrays of (nearly) identical size
numpy.hsplit(array, 3)	Split the array horizontally at 3rd index

– **More**

Operator	Description
other = ndarray.flatten()	Flattens a 2d array to 1d
array = np.transpose(other) array.T	Transpose array
inverse = np.linalg.inv(matrix)	Inverse of a given matrix

3. 6. Mathematics

– **Operations**

Operator	Description
np.add(x,y) x + y	Addition
np.subtract(x,y) x - y	Subtraction
np.divide(x,y) x / y	Division
np.multiply(x,y) x @ y	Multiplication
np.sqrt(x)	Square Root
np.sin(x)	Element-wise sine
np.cos(x)	Element-wise cosine
np.log(x)	Element-wise natural log
np.dot(x,y)	Dot product
np.roots([1,0,-4])	Roots of a given polynomial coefficients

– **Comparison**

Operator	Description
==	Equal
!=	Not equal
<	Smaller than
>	Greater than
<=	Smaller than or equal
>=	Greater than or equal
np.array_equal(x,y)	Array-wise comparison

– **Basic statistics**

Operator	Description
np.mean(array)	Mean
np.median(array)	Median
array.corrcoef()	Correlation Coefficient
np.std(array)	Standard Deviation

– **More**

Operator	Description
array.sum()	Array-wise sum
array.min()	Array-wise minimum value
array.max(axis=0)	Maximum value of specified axis
array.cumsum(axis=0)	Cumulative sum of specified axis

3. 7. Slicing and Subsetting

Operator	Description
array[i]	1d array at index i
array[i,j]	2d array at index[i][j]
array[i<4]	Boolean Indexing, see Tricks
array[0:3]	Select items of index 0, 1 and 2
array[0:2,1]	Select items of rows 0 and 1 at column 1
array[:1]	Select items of row 0 (equals array[0:1, :])
array[1:2, :]	Select items of row 1
[comment]: <> (array[1,...]
array[: -1]	Reverses array