Cairo University
Faculty of Computers and Artificial Intelligence
Machine Learning Course

Lab 1: Python

Lab Outline:

- ➢ Why use Python in this course?
- ➢ Environment setup
- ➢ Revising the basics of Python

Why Python?

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It is a fully functional programming language that can do anything almost any other language can do. It is used for machine learning for several reasons which include:

- **Simplicity:** Python is a friendly language where syntax is easy to read and follow.
- **Variety of libraries and frameworks:** Python has a full suite of tools for machine learning and data analytics (e.g. Scikit-learn, Keras, PyTorch, Pandas, etc.). Also, there is a Python package for most conceivable math functions such as:
  - *Numerical linear algebra: Numpy*
  - *General scientific computing (e.g. integration): Scipy*
  - *Statistical modeling: Statsmodel, PyMC3*
  - *Symbolic algebra: SymPy*

Environment Setup:

➔ Download the suitable Python version according to your OS from:

https://www.python.org/downloads/

➔ Use an IDE (like PyCharm, Spyder, etc.) to write complete Python programs.

Note: Python comes with a "Python shell" (and its IDLE) that you can use to run any Python command.

Basics of Python:

Primitive Data Types:

These are the most basic data structures. They are the building blocks for data manipulation and contain pure, simple values of data. Python has four primitive variable types:

- Integer
- Float
- String
- Boolean

In Python, we do not have to explicitly state the type of the variable because it is a dynamically typed language.

```
counter = 100          # An integer assignment
miles   = 1000.0       # A floating point
name    = "John"       # A string
a = b = c = 1
a, b, c = 1, 10.5, “hello”
```

Input and Output:

You can use **input()** to take user input:

```
>>> name = input("What is your name? ")
What is your name? Winston Smith
>>> name
'Winston Smith'
```

Note: The received data is always treated as string so you should cast it if you need another type.

You can use **print()** to print to screen:

```
>>> print(42)                                # <class 'int'>
42
>>> print(3.14)                              # <class 'float'>
3.14
>>> print(1 + 2j)                            # <class 'complex'>
(1+2j)
>>> print(True)                              # <class 'bool'>
True
```

```
>>> print('node', 'child', 'child', sep=' -> ')
node -> child -> child
```

```
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

Note: We can end a print statement with any character/string using the **"end"** *parameter.*

Casting:

```
>>> number = input("Enter a number: ")
Enter a number: 50
>>> print(number + 100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int

>>> number = int(input("Enter a number: "))
Enter a number: 50
>>> print(number + 100)
150
```

Note: We can use **type(<object>)** *to get the type of the object stored in any data type.*

Operators:

Type	Operators
Arithmetic operators	+, -, *, /, %, //, **
Comparison operators	>, <, ==, !=, >=, <=
Logical operators	and, or, not
Bitwise operators	&, |, ~, ^, >>, <<
Assignment operators	=, +=, -=, *=, /=, %=, //=, **=, &=, |=, ^=, >>=, <<=
Identity operators	is, is not
Membership operators	in, not in

Non-Primitive Data Structures:

Non-primitive types are the sophisticated members of the data structure family. They don't just store a value, but rather a collection of values in various formats. In Python, these types include:

- **List:** It is just like arrays declared in other languages. Lists don't need to be homogeneous.

  Example:

  my_list = [1, "name", True, 8.9, [ ]]

  Note: The main difference between a tuple and a list is that the tuple object is immutable. Once we've declared the contents of a tuple, we can't modify them.

- **Dictionary:** It is an unordered collection of data values used to store values like a map, so it holds a key:value pair.

  Example:

  my_dict = {'Name': 'John', 1: [1, 2, 3, 4]}

- **File:** Files are traditionally a part of data structures. The syntax to read and write files in Python is similar to other programming languages but a lot easier to handle. Some of the basic file functions include: **open(<filename>, <mode>)**, **read()**, **readline()**, **readlines()**, **write()** and **close()**.

Conditions:

```
>>> name = 'Joe'
>>> if name == 'Fred':
...     print('Hello Fred')
... elif name == 'Xander':
...     print('Hello Xander')
... elif name == 'Joe':
...     print('Hello Joe')
... elif name == 'Arnold':
...     print('Hello Arnold')
... else:
...     print("I don't know who you are!")
...
Hello Joe
```

Python does not use braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation which is rigidly enforced. The number of spaces in the indentation is variable but all statements within the block must be indented the same amount.

Conditions can also be written as follows:

<expr1> if <conditional_expr> else <expr2>

Loops:

In python, we can use while loops or for loops. Here's the syntax of a while loop:

while <conditional_expr>:

<body>

Here's the syntax of a for loop:

for <iterating_var> in <sequence>:

<body>

```
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if 'b' in i:
...         break
...     print(i)
...
foo
```

```
>>> x = range(5)
>>> x
range(0, 5)
>>> type(x)
<class 'range'>
```

```
>>> for n in x:
...     print(n)
...
0
1
2
3
4
```

Note: We can use a **break** *statement to exit any loop or a* **continue** *statement to skip iterations in the loop.*

Functions:

The usual syntax of functions in Python is:

def <function_name>([<parameters>]):

<statement(s)>

```
>>> def double(x):
...     return x * 2
...
```

Default parameters:

```
>>> def f(qty=6, item='bananas', price=1.74):
...     print(f'{qty} {item} cost ${price:.2f}')
...
```

```
>>> f(4, 'apples', 2.24)
4 apples cost $2.24
>>> f(4, 'apples')
4 apples cost $1.74

>>> f(4)
4 bananas cost $1.74
>>> f()
6 bananas cost $1.74

>>> f(item='kumquats', qty=9)
9 kumquats cost $1.74
>>> f(price=2.29)
6 bananas cost $2.29
```

Variable-length arguments:

```
>>> def f(*args):
...     print(args)
...     print(type(args), len(args))
...     for x in args:
...             print(x)
...

>>> f(1, 2, 3)
(1, 2, 3)
<class 'tuple'> 3
1
2
3
```

Lambda functions:

```
>>> (lambda x: x + 1)(2)
3
```

A lambda function is a small anonymous function. Lambda functions are efficient whenever we want to create a function that will only contain simple

expressions (expressions that are usually a single line of a statement) and we want to use this function once.

Practice:

Let's try to build a simple decision tree for classifying fruits into either watermelons or apples.

- Write a function **"classify"** that takes a dictionary containing a fruit's data in this format: **{"color":"green", "size":"small"}.** If the fruit's color is red, it is classified as an apple, otherwise the size is checked.

- Create a list of fruits:
  - **data = [{"color":"green", "size":"big"}, {"color":"orange", "shape":"round", "size":"big"}, {"color":"red", "size":"medium"}, {"color":"green", "size":"big"}, {"color":"red", "size":"small", "taste":"sweet"}, {"color":"green", "size":"small"}]**

- Use your function to classify each one (use different methods) and store the results in a list.
  - **results = [classify(x) for x in data]**
  - **results = list(map(classify,data))**
- Count the number of watermelons.
  - **sum([classify(x)=="watermelon" for x in data])**
- Count the number of unknown fruits using a different method.
  - **len(list(filter(lambda x: classify(x)=="unknown", data)))**