Lab 3

- **Package & import**
- **Classes**

1 Modules

Modular design requires a complex system to be divided into smaller, independent components called modules that are created and tested independently. A module is a file with the **".py" extension** that contains statements of variables, functions, classes, etc., that can be **imported inside another python program**.

A module is **loaded only once**, regardless of the number of times it is imported. This prevents the module execution from happening over and over again if multiple imports occur. Python searches for the modules in a list of directories defined in **sys.path**.

Ways to import modules:

1. the **import** statement - does not import the names of the functions defined in math directly into the current code. It only imports the module name.
   example: import math
2. the **from ... import** statement - we can import a specific function from a module without importing the whole module.
   example: from math import sqrt, pi
3. **import all function**
   example: from math import *
4. **import module as** - to avoid ambiguities you can import a module and assign it an alias.
   example: import math as mt

```
import sys
from math import sqrt
from os import *
import datetime as dt

>>> print(f"sys.version = {sys.version}")
sys.version = 3.8.10 (default, Jun 22 2022, 20:18:18) [GCC 9.4.0]

>>> print(f"sqrt(16) = {sqrt(16)}")
sqrt(16) = 4.0

>>> print(f"listdir() = {listdir()}")
listdir() = ['.idea', 'main.py', 'venv']

>>> print(f"dt.datetime.now() = {dt.datetime.now()}")
dt.datetime.now() = 2022-10-16 11:45:44.255425
```

1.1 Python Random Module

Python Random module is an in-built module of Python which is used to generate random numbers.
Example: Printing a random value from a list
Syntax: random.choice(sequence)

```
import random

list1 = [1, 2, 3, 4, 5, 6]
>>> print(random.choice(list1))
3
```

Example: Creating random integers
Syntax: randint(start, end)

```
import random

r1 = random.randint(5, 15)
>>> print("Random number between 5 and 15 is % s" % (r1))
Random number between 5 and 15 is 8

r2 = random.randint(-10, -2)
>>> print("Random number between -10 and -2 is % d" % (r2))
Random number between -10 and -2 is -5
```

Example: Creating random floats
Syntax: random.random()

```
from random import random

>>> print(random())
0.571793435556230
```

2 Packages

A Python **package** usually consists of **several modules**. Physically, a package is a **folder** containing modules and maybe other folders that themselves may contain more folders and modules. The path to the package must be included in the python sys.path (the same way as modules). Each package (directory) usually contains a file named **__init__().py**. The init file may be empty, or it may contain code that runs automatically when the package is imported, or when a module is imported from the package.

To import a specific module inside a package you must use the "." separator:
import dir1.dir2.my_module

Example of packages: scipy, pandas, Django, tensorflow, pytorch, matplotlib, seaborn, fastai, huggingface, scikit-learn, numpy

2.1 PIP

pip is a package-management system written in Python used to install and manage software packages, additional libraries and dependencies that are not distributed as part of the standard library. It connects to an online repository of public packages, called the Python Package Index.

```
# install pip
>>> sudo apt install python3-pip

>>> pip --version
pip 20.0.2 from /usr/lib/python3/dist-packages/pip (python 3.8)

>>> pip install some-package-name
```

```
>>> pip install --upgrade some-package-name

>>> pip uninstall some-package-name

# list installed packages
>>> pip list

# creates a list of all installed packages
>>> pip freeze > requirements.txt

# install full lists of packages and corresponding version numbers,
# through a "requirements.txt" file
>>> pip install -r requirements.txt
```

3 Python Classes and Objects

Create a Class:

Classes in Python are created using the **class** keyword. A class is a "blueprint" for creating objects. You can have several classes in one module (file). Pay attention to *self* - it represents the instance of the class.

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

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

3.1 Class Object

Class objects support two kinds of operations: **attribute references** and **instantiation**.

Attribute reference:

Attribute references use the standard syntax used for all attribute references in Python: **obj.name**. In the example above, **MyClass.i** returns an integer and **MyClass.f** returns a function object.

```
>>> print(MyClass.i)
12345
>>> print(type(MyClass.f))
<class 'function'>
```

Class instantiation:

Class instantiation uses function notation.

```
x = MyClass()  # create an object
del x          # delete the object
```

The operation above creates an empty object. To add initial properties to the object, a class may define a special method named **__init__()**. This is the equivalent of a constructor in Java. The **__init__()** function is called automatically every time the class is being used to create a new object.

```
class Person:
    def __init__(self, name, age):
        self.name = name
```

```
        self.age = age


    def f(self):
        return "I'm character from tv series Lost."


p1 = Person("James", 39)  # create an object

print(p1.name)
print(p1.age)
```

3.2 Instance Objects

The only operations understood by instance objects are **attribute references**. There are two kinds of valid attribute names: **data attributes** and **methods**.

Data attributes:

```
p1.job = "salesman"  # adding a new attribute
p1.name = "Sawyer"   # modifying an attribute
print(p1.__dict__)   # prints the attributes and the values
print(vars(p1))      # same as the line above
del p1.job           # deleting the object attribute
```

Methods:

A method is a function that "belongs to" an object. So, for the Person class, **p1.f** is a valid method reference, since **Person.f** is a function.

```
Person.f  # returns a function object
p1.f      # returns a method object
```

Methods are functions with the instance object as first argument of the function. The instance object is referred as the **self** parameter in the method. The **self** keyword is equivalent with the **this** keyword in Java. The following calls are exactly the same:

```
Person.f(p1)  # calling the function
p1.f()        # calling the method
```

3.3 Class and Instance Variables

Instance variables are attributes and methods unique to each objects while **class variables** are attributes and methods shared by all instances.

```
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'
```

Note: shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries.

```
class Dog:
    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

3.4 Inheritance

Inheritance is a core concept in object-oriented programming (OOP) paradigm. It is a mechanism in which one class inherits the attributes and methods of another class. The syntax looks as follows:

```
class ChildClass(ParentClass):
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname
    ...

# if the parent class is in another module
class ChildClass(module_name.ParentClass):
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname
    ...
```

Note: The child's **__init__()** function overrides the inherited one. To keep the inheritance of the parent's **__init__()** function, add a call to the parent's **__init__()** function either by calling it using the parent class name:

```
class ChildClass(ParentClass):
    def __init__(self, fname, lname):
        ParentClass.__init__(self, fname, lname)  inheriting properties
        self.age = 39  # adding new property to the Child class
    ...
```

or by using the **super** function:

```
class ChildClass(ParentClass):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)  # inheriting properties
        self.age = 39  # adding new property to the Child class
    ...
```

Functions that work with inheritance:

isinstance(object, classinfo)	Return True if the object argument is an instance of the classinfo argument otherwise return False
issubclass(class, classinfo)	Return True if class is a subclass of classinfo otherwise return False. A class is considered a subclass of itself.

```
class Animal:
    pass

class Dog(Animal):
    pass

class Student:
    pass

animal = Animal()
dog = Dog()
student = Student()

>>> print(isinstance(animal, Animal))
True
>>> print(isinstance(animal, Dog))
False
>>> print(isinstance(dog, Animal))
True
>>> print(isinstance(dog, Dog))
True
>>> print(isinstance(student, Animal))
False

>>> print(issubclass(Animal, Animal))
True
>>> print(issubclass(Animal, Dog))
False
>>> print(issubclass(Dog, Animal))
True
>>> print(issubclass(Dog, Dog))
True
```

3.4.1 Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class ChildClass(Parent1, Parent2, Parent3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

If an attribute is not found in ChildClass, it is searched for in Parent1, then (recursively) in the parent classes of Parent1, and if it was not found there, it was searched for in Parent2, and so on.

3.5 Private Variables

"Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a mechanism, called **name mangling**, that allows you to achieve something

similar. Any identifier of the form **__ia23** (at least two leading underscores, at most one trailing underscore) is textually replaced with **_classname__ia23**, where classname is the current class name with leading underscore(s) stripped.

```
class MyClass:
    a = 1
    __b = 20  # private variable

    def __init__(self, c, d):
        self.c = c
        self.__d = d  # private variable

x = MyClass(99,100)

>>> print(x.a)
1

>>> print(x.__b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__b'

>>> print(x._MyClass__b)
20
```

In addition, Python has a convention that names that are prefixed with an underscore (e.g. **_ia23**) should be treated as "private" and not be used outside of the class (or module) even if you're not technically prevented from calling it from somewhere else.

So, as a general rule in Python: You don't play around with another class's variables that look like **__ia23** or **_ia23**.

4 Exercises

1. Implement the softmax function.

$$softmax(X)_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

   You can use *exp* function from *math* package. Compare your results with the result from scipy.special.softmax

```
a=[-0.4,0, 0.2, 0.2, 0.4]
>>> softmax(a)
array([0.11959429, 0.17841372, 0.217915  , 0.217915  , 0.26616199])
```

2. Download the archive lab3.tgz and extract the content into a folder. Open the folder with your IDE and do the following exercises:

   - Add a method for FruitShop in shop.py script that returns a list of all the fruits sold in that shop. Use this method on the objects instantiated in *shopSmart.py* (*shop*1 and *shop*2) and print the result.
   - We want to be able to print the list of all the fruits which are sold in a shop every time we print the object. Identify the required method and change it; test it in shopSmart.py.
   - Define the equality between two shops like this: two shops are considered equal if they sell the same fruits (no matter the price). Overwrite the *__eq__* method and test *shop*1 == *shop*2 before and after your change.

     HINT

```
#Python automatically calls the __eq__ method of a class when
#you use the == operator to compare the instances of the class.

class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.age == other.age

        return False

# so now two persons are considered equal
# if they have the same age
```

- Fill in the function *shopSmart*(*orderList*, *fruitShops*) in *shopSmart.py*, which takes an orderList and a list of FruitShop and returns the FruitShop where your order costs the least amount in total.

3. Let's play with graphs.

   - Consider you have a directed graph stored in a file having the following structure:

     ```
     verticesNumber
     node1 node2
     node1 node3
     ....
     ```

     On the first line, you have the total number of vertices in the graph. From the second line, on each line, you have an edge (in the given example, there is an edge from node1 to node 2 and to node 3; the separator is space). Read this file and create a graph, where a graph is represented as a list of Nodes.
     HINT

```
class Node():

    def __init__(self, vertex_number):
        self.vertex_number = vertex_number
        self.neighbours = []

    def edge(self, node):
        self.neighbours.append(node)

    def __str__(self):
        return f'({self.vertex_number})'

#example to create a node
node1 = Node(1)
node1.edge(2) #adds the edge from 1 to 2
#example to create a graph
nodes = []
for i in range(7):
    nodes.append(Node(i))
nodes[1].edge(nodes[3])
nodes[1].edge(nodes[6])
nodes[2].edge(nodes[6])
```

```
nodes[3].edge(nodes[3])
```

- Implement a function to print the graph (use the string representation of Node objects): it prints all the edges in the graph, each edge on different line.
- Create a method that returns the out-degree of that node and use it to find the vertex with the highest out-degree. For example, in the graph from the HINT, the out-degree of the vertex 1 is 2.
- Create a subgraph with a random selection of vertices(60% of the initial number) and random selection of edges for each vertex (80% of the initial number of neighbors for each node).

  HINT: use random.choices

In case you want to plot the graphs, you can use for example networkx, matplotlib, and numpy like this:

```
>>> import numpy as np
>>> import networkx as nx
>>> import matplotlib.pyplot as plt
>>>
>>> A = np.matrix([[1,1],[2,1]])
>>> G = nx.from_numpy_matrix(A)
>>> nx.draw(G)
>>> plt.show()
```