# Fundamentals of Programming – Final exam (example)

**Important:** Provide the best possible solutions in each exercise. In addition to working properly, procedures and functions must be well programmed (using the most appropriate instructions, without unnecessary operations or variables, etc.).

**Exercise 1 (3'5 points)**

Define the `Country` class with the following attributes: the name of the country, its area (km²) and a list of the ministers of its government. Then:

a) Define the member functions required to initialize the attributes and the corresponding `getters` and `setters` attribute values. If when initializing the class, the user does not pass the parameter list of ministers, initialize the list with an empty list. **(0'5 points)**
b) Define the following member methods: **(0'5 points)**

- `__str__` to print objects of the class `Country` with the format:
  ```
  <name country> (area: <area>)
      Government: <name_minister_1>, ..., <name_minister_N>
  ```
- operator `__ge__` (greater or equal >= ) to compare two countries. One country is "bigger or equal" than another if its area is greater or equal than the other.
- `PushMinister(m)` to add a minister `m` to the list of ministers.
- `PopMinister(m)` to remove a minister `m` from the list of ministers. Make sure the element to be deleted is in the list, otherwise, print an error message.

  The parameter `m` in `PushMinister` and `PopMinister` is a string: the minister's name.

```
class Country():
    def __init__(self,c_name,c_area,l_ministers=[]):
        self.name = c_name
        self.area = c_area
        self.ministers = l_ministers

    def get_name(self):
        return self.name

    def set_name(self,n):
        self.name = n

    def get_area(self):
        return self.area

    def set_area(self,a):
        self.area = a

    def get_ministers(self):
        return self.ministers

    def set_ministers(self,l):
        self.ministers = l
```

```
    def __str__(self):
        s = self.name
        s += " (area: " + str(self.area) + ") "
        s += " Government: "+ ", ".join(self.ministers)
        return(s)

    def __ge__(self,other):
        return(self.area >=other.area)

    def PushMinister(self,m):
        self.ministers.append(m)

    def PopMinister(self,m):
        if m in self.ministers:
            self.ministers.remove(m)
        else:
            print("Error: Minister does not exist")
```

c) Modify the previous code to use *property decorators* instead. **(0'5 points)**

```
class Country():
    def __init__(self,c_name,c_area,l_ministers=[]):
        self._name = c_name
        self._area = c_area
        self._ministers = l_ministers

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self,n):
        self._name = n

    @property
    def area(self):
        return self._area

    @area.setter
    def area(self,a):
        self._area = a

    @property
    def ministers(self):
        return self._ministers

    @ministers.setter
    def ministers(self,l):
        self._ministers = l

    def __str__(self):
        s = self.name
        s += " (area: " + str(self.area) + ") "
        s += " Government: "+ ", ".join(self.ministers)
        return(s)
    def __ge__(self,other):
        return(self.area >=other.area)

    def PushMinister(self,m):
        self.ministers.append(m)

    def PopMinister(self,m):
        if m in self.ministers:
            self.ministers.remove(m)
        else:
            print("Error: Minister does not exist")
```

d) Make a main program that asks the user the name of a file (containing info of countries), reads the file contents, and creates a dictionary of countries. In the file, there is one country per row, where the information is separated using commas. For example: **(0’75 points)**

```
<country1>,<area1>,<minister1>,<minister2>,<minister3>,<minister4>
<country2>,<area2>,<ministerA>
<country3>,<area3>,<ministerX>,<ministerY>,<ministerZ>
```

Use exceptions to control that the file can be opened. In case of an error, manage the exception `IOError` and print the error message “Error: file not found”.

In the dictionary, the key is the name of the country. The values will be another dictionary with the area and the list of ministers, as follows:

DCountries[<name_country>] = {‘Area’:<area_country>, ‘Ministers’:[<m1>,<m2>,…]}

```
nf = input("Name file:")
try:
    f = open(nf,'r')
    dcountries= {}
    for tline in f:
        t=tline[:-1]
        s=t.split(",")
        lmin = s[2:len(s)]
        dcountries[s[0]] = {"Area":int(s[1]),"Ministers":lmin}
    f.close()
    print(dcountries)

except IOError:
    print("Error: file not found")
```

e) Make the function `GreaterCountries` that receives the dictionary of countries created above, asks the user an area A, and returns another dictionary with countries with an area greater than A (in km$^2$). **Do not** use **dictionary comprehension**. **(0’5 points)**

```
def GreaterCountries(dc):
    val = int(input("Write minimum area:"))
    dc2 = {}
    for k,v in dc.items():
        if(dc[k]['Area']>val):
            dc2[k] = v
    return dc2
```

f) Write the previous function, but now use **dictionary comprehension. (0’75 points)**

```
def GreaterCountries(dc):
    val = int(input("Write minimum area:"))
    dc2 = {k:v for (k,v) in dc.items() if v['Area']>val}
    return dc2
```

**Exercise 2 (1 point)**
Write the function `members(x, y, z, pos)` that will receive as parameters: x: list of names, y: list of genders, z: list of birthyears, pos: list of indices. These lists contain information about athletes in a sport club. The list of indices tells at which position the members in each category start. There are 3 categories. Suppose that the lists are ordered as follows:

- Athletes in category 1: from the beginning until pos[0] (exclusive)
- Athletes in category 2: from pos[0] (inclusive) until pos[1] (exclusive)
- Athletes in category 3: from pos[1] (inclusive) until the end

The program will return a list of strings. For example, if x = ['Joan', ‘Nuria’, ‘Lucas’,'Miquel'], y = ['M’, 'F', 'M', 'M'], z = [1995, 2005, 2001, 2008], pos = [2,3], the output of the function should be:

['Cat 1: Joan, M, 1995', 'Cat 1: Nuria, F, 2005', 'Cat 2: Lucas, M, 2001', 'Cat 3: Miquel, 2008']

IMPORTANT. The solution is only considered CORRECT if it uses **zip** and **enumerate**.

```
def members(x, y, z, pos):
    res=[]
    for a,b in enumerate(zip(x, y, z)):
        print('-----',a)
        if 0 <= a < pos[0]:
            t="Cat 1: "
        elif pos[0] <= a < pos[1]:
            t="Cat 2: "
        else:
            t="Cat 3: "
        res.append(t+b[0]+", "+b[1]+", "+str(b[2]))
    return res
```

**Exercise 3 (1 point)**
How long is a year in Earth on other planets? Write a function that computes the equivalence of Earth years into years in each planet of the solar system. The function will receive, as a parameter, a value in seconds, and the list of equivalences with the solar planets. It will return a list with the equivalences on each planet.

One year on Earth (365.25 days) corresponds to 31,557,600 seconds. The equivalences with the other planets are:

- Mercury: 1 year = 0.24 Earth years.
- Venus: 1 any = 0.62 Earth years.
- Mars: 1 any = 1.88 Earth years.
- Jupiter: 1 any = 11.86 Earth years.
- Saturn: 1 any = 29.45 Earth years.
- Uranus: 1 any = 84.02 Earth years.
- Neptune: 1 any = 164.79 Earth years.

For example, if the function receives the value of 1,000,000,000 (Earth) seconds and the list of equivalences [0.24, 0.62, 1, 1.88, 11.86, 29.45, 84.02, 164.79], the function should return the list:

[132.03370, 51.10982, 31.68808, 16.85537, 2.67185, 1.07600, 0.37715, 0.19229]

This means that 1,000,000,000 seconds are 132 years on Mercury, 51 years on Venus, etc.

Complete the instructions in red color:

```
def conversion(earthseconds,listequiv):

    l = list(map(lambda x: (earthseconds/31557600)/x, listequiv))

    return l


#MAIN
equiv=[0.24, 0.62, 1, 1.88, 11.86, 29.45, 84.02, 164.79]
res= conversion(1000000000,equiv)
print(res)
```

**IMPORTANT:** The SOLUTION will only be considered CORRECT if it uses **`map`** and **`lambda`**.

**Exercise 4 (1 point)**
Write a function called *intersection* that receives two lists as parameters and returns a list that is the intersection (repeated elements) of the two lists.

a) Write the function without list comprehension

```
def intersection1(l1, l2):
    l3=[]
    for i in l1:
        if i in l2:
            l3.append(i)
    return l3
```

b) Write the function using list comprehension

```
def intersection(l1, l2):
     l3 = [value for value in l1 if value in l2]
     return l3
```

**Exercise 5 (1 point)**

Using `numpy` package, write a function that returns the sum of all values that do not belong to the diagonal of a square matrix. For this, follow these steps:

- Ask the user to introduce the integer value N. Then, create a matrix **`m`** of size NxN (a square matrix) and initialize it with random float values between [0,10].
- Create a square matrix of Booleans **`noDiagonal`** (initialized with 1's) where the diagonal elements are set to 0's. This can be seen as a mask (only cells with 1 are considered).
- Multiply **`m`** with **`noDiagonal`** (element-wise) and sum the elements to return the total sum.

Complete the code:

```
def sumNoDiagonal():
    N = int(input("N:"))
    m = np.random.rand(N,N)
    m = m*10

    noDiagonal = np.ones(m.shape)
    noDiagonal[np.arange(0,N,1), np.arange(0,N,1)] = 0
    msum = (m * noDiagonal).sum()
    return msum
```

**Exercise 6 (1 point)**

Define an abstract class `Animal` and the subclasses `Human`, `Fish`, `Bird`. The class `Animal` will have an abstract method called `move`. The subclass Human will have the implementation of this method as follows:

- The method `move` of the class `Human` prints the string “Humans can walk”.
- The method `move` of the class `Fish` prints the string “Fish can swim”.
- The method `move` of the class `Bird` prints the string “Birds can fly”.

If the abstract method `move` is called, it should raise `NotImplementedError().`

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        raise NotImplementedError()

class Human(Animal):
    def move(self):
        print("Humans can walk")

class Fish(Animal):
    def move(self):
        print("Fish can swim")

class Bird(Animal):
    def move(self):
        print("Birds can fly")
```

**Exercise 7 (1’5 points)**

Implement a **recursive** function that inverts the elements in a list. For example, if the input list is [4,8,12,20,5,2], then the output list should be [2,5,20,12,8,4].

Complete the code:

```
def invertListRec(l,i,j):

    if(i>=j):
        return (l,i,j)
    else:
        (l[i],l[j]) = (l[j],l[i])
        invertListRec(l,i+1,j-1)
        return (l,i,j)


#MAIN
l = [4,8,12,20,5,2,0]
i = 0
j = len(l)-1
(l2, i, j) = invertListRec(l,i,j)
print(l2)
```

**IMPORTANT:** The SOLUTION will only be considered CORRECT if it is recursive.