Fundamentals of Programming – **Final exam** (January 12, 2022)

Surnames, Name: **NIU:**

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 (5 points)

We need to manage an online shop that sells electronic books and music. Customers can browse the catalog and buy each one of the products (either ebook or mp3), which will be send to the customer's email.

a) **(0'5 points).** Define the `Item` class with the following attributes: `code` (a string, the code of the product), and `prize` (an integer, the cost of the product). Implement the `init` method to initialize the attributes. Use *property decorators*.

```
class Item:
     def __init__(self, code, prize):
          self._code = code
          self._prize = prize

     @property
     def code(self):
          return self._code
     @code.setter
     def code(self, value):
          self._code = value
     @property
     def prize(self):
          return self._prize
     @prize.setter
     def prize(self, value):
          self._prize = value
```

b) **(0'5 points).** Define the `Book` class as a subclass of `Item`. In addition to the attributes of Item, it also has the following strings as attributes: `title`, `author`. Implement the `init` method as follows: call the `init` method of the superclass to initialize `code` and `prize`, and then initialize the rest of the attributes in `Book`. In case no parameters are passed, initialize them by default with *code*="", *prize*=0, *title*="", *author*="". Also implement the method `info`, which returns a string with the format `(<code>;<prize>;<title>;<author>)`. Use getters/setters or *property decorators.*

```
class Book(Item):

    def __init__(self, code="", prize = 0, title="", author=""):
        super().__init__(code,prize)
        self._title = title
        self._author = author
    @property
    def title(self):
        return self._title
    @title.setter
    def title(self, value):
        self._title = value
    @property
    def author(self):
        return self._author
    @author.setter
    def author(self, value):
        self._author = value

    def info(self):
        return (self.code + ";" + str(self.prize) + ";"+ self.title + ";"
+ self.author)
```

c) **(0'5 points).** Define the `Music` class as a subclass of `Item`. In addition to the attributes of Item, it also has the following attributes: `song` (string) and `artists` (a set of strings, with correspond to the musicians that play in the song). Implement the `init` method as follows: call the `init` method of the superclass to initialize `code` and `prize`, and then initialize the rest of the attributes in `Music`. In case no parameters are passed, initialize them by default with *code*="", *prize*=0, *song*="", and *artists* as empty set. Also implement the method `info`, which returns a string with the format `(<code>;<prize>;<song>;<artists>)`. Use getters/setters or *property decorators.*

```
class Music(Item):

    def __init__(self, code="", prize = 0, song="", artists=set()):
        super().__init__(code,prize)
        self._song = song
        self._artists = artists

    @property
    def song(self):
        return self._song
    @song.setter
    def album(self, value):
        self._song = value
    @property
    def artists(self):
        return self._artists
    @artists.setter
    def artists(self, value):
        self._artists = value

    def info(self):
        return (self.code + ";" + self.prize + ";"+ self.song + ";" +
";".join(list(self.artists)))
```

d) **(1'25 point).** Implement the function `read_items(fn)` that receives, as input, a filename that contains, in each row, the information of each item. The first value of the line indicates the type of element: if it is a book ('B') or music ('M'). The rest of the information (code, prize,…) is separated with commas. For example:

```
B,SF10,5,I robot,Asimov
M,AZ20,2,November Rain,Axl Rose,Izzy Stradlin
B,YM11,8,Foundation,Asimov
M,SW21,3,Welcome to the jungle,Axl Rose,Izzy Stradlin,Slash
```

The function will read the file and return *LItems,* a list of objects of class Book or Music. Follow these steps:

1. Create the empty list *LItems*.
2. Open the file. Use exceptions to control that the file can be opened. In case of an error, manage the exception IOError and print the message "Error: file not found".
3. If type "B", create an object of type Book, initialize it and add it to the *LItems* list.
4. If type "M", create an object of type Music, initialize it and add it to the *LItems* list.
5. If type is none of these, print an error message.
6. Once all file is read, return the list.

```
def read_items(name_file):
    LItems = []
    try:
        f=open(name_file,"r")
        for tline in f:
            lin = tline[:-1]
            v = lin.split(",")
            if v[0] == 'B':
                b = Book(v[1], int(v[2]),v[3], v[4])
                LItems.append(b)
            elif v[0] == 'M':
                art = set()
                for a in range(4,len(v)):
                    art.add(v[a])
                m = Music(v[1],int(v[2]),v[3],art)
                LItems.append(m)
            else:
                print('Error, type of product must be B or M')
        f.close()
    except IOError:
        print('Error: file not found')
    return LItems
```

e) **(1 point).** Implement the `invert_dictionary(DSales)` function that receives as a parameter a dictionary `DSales`, where the keys are the codes of customers (e.g. DNIs), and their associated values corresponds to the list of codes of products that each customer has bought. The function will return the dictionary `DProducts`, where the keys are the codes of the products, and the values correspond to the list of users that bought this product. For example:

DSales

Usercode	Products
U1	A5,B2,B7
U2	B7,B2
U3	B2

$\rightarrow$

DProducts

Product	Usercodes
A5	U1
B2	U1,U2,U3
B7	U1,U2

```
def invert_dictionary(DSales):

    DProducts=dict()
    for (user,listproducts) in DSales.items():
        for product in listproducts:
            if product not in DProducts:
                l = [user]
                DProducts[product]= l
            else:
                DProducts[product].append(user)

    return DProducts
```

f) **(1'25 points)** Implement the `top_products(DProducts)` function that receives a dictionary of products (key = code product, values = list of users that bought the product), and returns a dictionary `TopProd` with those very popular products. If the top product has been sold X times, popular products are those that have been sold a minimum of X/2 times. Follow these steps:

- Line 1. Create the list `LNumSold` with the number of units that have been sold for each product. Use **map** and **lambda**.
- Line 3. From `DProducts`, create the dictionary `TopProd` with only those elements in which the number of units sold >= *most*. Use **dictionary comprehension**.

```
def top_products(DProducts):

    1. LNumSold = list(map(lambda x:len(x),DProducts.values()))

    2. most = max(LNumSold) / 2

    3. TopProd = {k:v for (k,v) in DProducts.items() if len(v)>=most}

    4. return TopProd
```

Exercise 2 (1 point)

Write the generator *power2*(x) that returns the first x numbers of 2^n. Use the YIELD function.

```
def power2(x):
        yield 1
        i = 1
        r = 1
        while i<x:
            r= r * 2
            i +=1
            yield r
```

Exercise 3 (2 points)

Implement a **recursive** function GCD that receives two integers and computes its GCD - *Greatest Common Divisor* (“máximo común divisor”) using Euclides’ algorithm. The GCD is the largest positive integer that divides each of the integers (for example, the GCD of 8 and 12 is 4).

Euclides' algorithm consists of subtracting the smallest number from the highest until two equal numbers remain, which will be the greatest common divisor of the two initial numbers. For example, if the two numbers are 96 and 44, Euclides' algorithm would do:

96	52	8	8	8	8	8	4	**4**
44	44	44	36	28	20	12	8	**4**

Then make the main program ask the user two integers N1 and N2, call the GCD function and print the message: "GCD(N1,N2)= M" where M will be the output of the GCD function.

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

```
def GCD(n1,n2):
    if n1==n2:
        return n1
    elif n1>n2:
        return GCD(n1-n2,n2)
    else:
        return GCD(n1,n2-n1)

#MAIN
N1 = int(input("N1="))
N2 = int(input("N2="))
print("GCD("+str(N1)+","+str(N2)+")=",GCD(N1,N2))
```

Exercise 4 (2 points)
Write the output of these programs.

a)

Code:

```
import numpy as np
a=np.arange(5,65,5).reshape(3,4)
print(a)
a[1:3,:]+=20
print(a)
a[:,3]=a[:,0]-5
print(a)
```

Output:

```
[[ 5 10 15 20]
 [25 30 35 40]
 [45 50 55 60]]

[[ 5 10 15 20]
 [45 50 55 60]
 [65 70 75 80]]

[[ 5 10 15  0]
 [45 50 55 40]
 [65 70 75 60]]
```

b)

Code:

```
print([[x,y] for x,y in enumerate([x for x in range(5,20,3) if x%2==1])])
```

Output:

```
[[0, 5], [1, 11], [2, 17]]
```

c)

Code:

```
from abc import ABC, abstractmethod

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

class Pinguin(Animal):
    def __init__(self, n):
        self.name = n

class Lion(Animal):
    def __init__(self, n):
        self.name = n

    def eat(self):
        print("Lions eat meat")

#MAIN
l = Lion("Simba")
l.eat()
p = Pinguin("Pingu")
print(p.name)
```

Output:

```
Lions eat meat
```

ERROR (method **eat** is not implemented):

```
    TypeError: Can't instantiate abstract class Pinguin with abstract methods eat
```

d)

Code:

```
pron = ['me', 'him', 'them']
verb = ['pay', 'take']
f = [v +' '+p for p in pron for v in verb]
print(f)
```

Output:

```
['pay me', 'take me', 'pay him', 'take him', 'pay them', 'take them']
```