Sets and maps

Hello students,
during last lecture you learned about sets and maps. Today, we will complete some of the examples provided by the reference-book.

Assignment 1 - Sets

Assignment 1.0 - Transcribe the example

In the reference-book, is proposed a basic partial implementation of the **Set ADS**. This is shown in the listing **3.1** (**pages 73 - 74**).

Read through the code, and copy it to a file called "linearset.py".

ATTENTION : there may be a few mistakes in the code of the book, fix them!

Assignment 1.1 - Complete the example

Some methods of the aforementioned class were not implemented.

Implement the missing methods of the Set class:

- **`intersect`**`(setB)` - Creates and returns a new set that is the intersection of this set and setB. The intersection of sets A and B contains only those elements that are in both A and B. Neither set A nor set B is modified by this operation.
- **`difference`**`(setB)` - Creates and returns a new set that is the difference of this set and setB. The set difference, A − B, contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation

Assignment 1.2 - Expand the constructor

Modify the Set() constructor to accept an optional variable argument to which a collection of initial values can be passed to initialize the set. The prototype for the new constructor should look as follows:

```
def Set( self, *initElements)
```

It can then be used as shown here to create a set initialized with the given values:

```
>>> s = Set( 150, 23, 23, 86, 49 )
>>> len(s)
4
```

Assignment 1.3 - proper subset

Implement the method "**`is_proper_subset(setB)`**". The definition of **proper subset** is:

> *Given two sets, A and B, A is a proper subset of B, if A is a subset of B and A does not equal B.*

Example of use:

```
>>> setA = Set(2, 0, 1)
>>> setB = Set(2, 0, 1)
>>> setC = Set(2, 0)

>>> setA.isSubsetOf(setB)
True

>>> setA.is_proper_subset(setB)
False

>>> setC.is_proper_subset(setA)
True
```

Assignment 1.4 - make it printable

Add the **`__str__`**() method to the Set implementation to allow a user to print the contents of the set. The resulting string should look similar to that of a list, except you are to use curly braces to surround the elements.

E.g. (the print function implicitly calls __str__):

```
>>> setA = Set(2, 0, 1)
>>> print(setA)
{2, 0, 1}
```

Assignment 1.5 - implement magic operators

Add Python's magic operator methods to the Set class that can be used to perform similar operations to those already defined by named methods:

Magic Method	Current Method
`__add__(setB)`	`union(setB)`
`__mul__(setB)`	`intersect(setB)`
`__sub__(setB)`	`difference(setB)`
`__lt__(setB)`	`isSubsetOf(setB)`

Assignment 1.6 - adding the iterator

Design and implement the iterator class **`_SetIterator`** for use with the **Set ADT** implemented using a list.

Example:

```
>>> setA = Set(2, 0, 1)
>>> for e in setA:
>>>     print(f"{e} ", end="")
2 0 1
```

Assignment 2 - Maps

Assignment 2.0 - Transcribe the example

In the reference-book, is proposed a basic partial implementation of the **Map ADS**. This is shown in the listing **3.2** (**pages 78 - 79**).

Read through the code, and copy it to a file called “linearmap .py”.

ATTENTION : there may be a few mistakes in the code of the book, fix them!

Assignment 2.1 - Get the keys

Add a new operation **`keyArray`**`()` to the Map class that returns an array containing all of the keys stored in the map. The array of keys should be in no particular ordering.

Example:

```
>>> m = Map()
>>> m.add("foo", 1)
>>> m.add("bar", 3)
>>> m.add("baz", 3)

>>> m.keyArray()
['foo', 'bar', 'baz']
```

Assignment 2.2 - implement magic operators

Add Python’s magic operator methods to the Set class that can be used to perform similar operations to those already defined by named methods:

Magic Method	**Current Method**
`__setitem __(ley, value )`	`add (key, value)`
`__getitem__(key )`	`valueOf(key)`

Assignment 2.3 - adding the iterator

Design and implement the iterator class **`_MapIterator`** for use with the **Map ADT** implemented using a list.

Example:

```
>>> m = Map()
>>> m.add("foo", 1)
>>> m.add("bar", 3)
>>> m.add("baz", 3)
>>> for entry in m:
>>>     print(f"{entry.key} - {entry.value}", end="\t")
foo - 1         bar - 3         baz - 3
```

Assignment 3 - multidimensional array

Assignment 3.0 - Transcribe the example

In the reference-book, is proposed a basic partial implementation of the **MultiArray ADS**. This is shown in the listing **3.3** (**pages 86 - 87**).

Read through the code, and copy it to a file called “multiarray.py”.

ATTENTION : there may be a few mistakes in the code of the book, fix them!

Assignment 3.1 - Implement the missing methods

During the lecture and in section 3.3 of the book, it was explained a common implementation of a multi-dimensional array using an efficient linear array.

In this exercise you are asked to complete the missing methods to correctly use the elements of this array.

We saw that the generic equation to locate the correct element is the following:

$$index_n(i_1, i_2, \ldots, i_n) = i_1 \cdot \prod_{k=2}^{n} d_k + i_2 + i_2 \cdot \prod_{k=3}^{n} d_k + i_2 + \ldots + i_n$$

Which can be abstracted in:

$$index_n(i_1, i_2, \ldots, i_n) = \; i_1 \cdot f_1 + \; i_2 \cdot f_2 + \ldots + \; i_n \cdot f_n$$

Where the factors are defined as:

$$f_1 = 1 \text{ and } f_i = \prod_{k=i+1}^{n} d_k$$

The methods you are asked to implement are:

- **`_computeFactors`**(*dimensions) - This method is called when creating a new instance (i.e. when calling `__init__`). It returns an Array containing the aforementioned factors $f_1, f_2, \ldots, f_n$. These should be stored as an instance variable.
- **`_computeIndex`**() - Given the factors and the tuple provided by the user (e.g. for accessing element at (0, 2, 1) in a 3-dimensional array, returns the correct index to find the element.

Example:

```
>>> m = MultiArray(3, 3, 3)
>>> m[0, 2, 1] = 42
>>> m[0, 2, 1]
42
```