A Python Framework for Neutrosophic Sets and Mappings

Giorgio Nordo[1,*], Saeid Jafari[2], Arif Mehmood[3] and Bhimraj Basumatary[4]

[1]MIFT Department – Mathematical and Computer Science, Physical Sciences and Earth Sciences University of Messina, 98166 Sant'Agata, Messina, Italy; giorgio.nordo@unime.it

[2]Mathematical and Physical Science Foundation, 4200 Slagelse, Denmark; jafaripersia@gmail.com

[3]Department of Mathematics, Institute of Numerical Sciences, Gomal University, Dera Ismail Khan 29050, KPK, Pakistan; mehdaniyal@gmail.com

[4]Department of Mathematical Sciences, Bodoland University, Kokrajhar, BTAD, India; brbasumatary14@gmail.com

*Correspondence: giorgio.nordo@unime.it

Abstract. In this paper we present an open source framework developed in Python and consisting of three distinct classes designed to manipulate in a simple and intuitive way both symbolic representations of neutrosophic sets over universes of various types as well as mappings between them. The capabilities offered by this framework extend and generalize previous attempts to provide software solutions to the manipulation of neutrosophic sets such as those proposed by Salama et al. [21], Saranya et al. [23], El-Ghareeb [7], Topal et al. [29] and Sleem [26]. The code is described in detail and many examples and use cases are also provided.

Keywords: neutrosophic set; neutrosophic mapping; Python; class; framework.

———

1. Introduction

Since the notion of neutrosophic set was introduced in 1999 by Smarandache [27] as a generalization of both the notions of fuzzy set introduced by Zadeh [31] in 1965 and intuitionistic fuzzy set proposed by Atanassov [4] in 1983, neutrosophic set theory had a rapid development and has been profitably used in many fields of pure Mathematics [1, 12, 13, 15, 16] as well as in several areas of applied sciences such as Graph Theory [5], Decision Making [14], Medicine [6], Statistics [10, 24, 28], Image Analysis [9, 32] Machine Learning [8, 25], etc.

In numerous instances, especially when dealing with applications stemming from real-world issues, manually manipulating neutrosophic sets that possess a finite yet consistent number of elements, along with their associated mappings, can be quite laborious and challenging. Consequently, there exists a significant demand for a system that can streamline the automation

———

of key neutrosophic operations, including union, intersection, neutrosophic difference, and the calculation of neutrosophic images or counterimages by mappings. Previous attempts to address this need were undertaken by Salama et al., who initially employed tools like Microsoft Excel [20] and later transitioned to using C# [21]. Another software application for processing neutrosophic sets developed in C# was described by Saranya et al. [23]. More recently, El-Ghareeb introduced a Python package designed to handle both single and interval-valued neutrosophic numbers and sets [7]. Unfortunately, however, in the latter paper the two classes concerning neutrosophic sets are described incompletely and summarily than those concerning neutrosophic numbers. Version 0.0.5 of this software that we consulted does not appear to provide adequate functionality even for the main neutrosophic operations and in any case the related repository on GitHub of the source code mentioned in the article does not appear to be available. Furthermore, other Python-based software solutions for handling neutrosophic numbers and matrices have been proposed by Topal et al. [29] and Sleem [26]. However, the authors are not aware of any other Python software specifically designed for the manipulation of neutrosophic sets is currently known.

This underscores the ongoing requirement for a set of well-structured Python classes, ideally available under an Open Source license, that enable automated and interactive manipulation of symbolic representations of neutrosophic sets, along with their associated mappings. Additionally, there is a need for comprehensive documentation and user-friendly design to facilitate straightforward integration for future implementations.

For this reason, we intended to design and develop a modern framework that extends and generalizes the above software solutions overcoming some of their limitations and offering greater flexibility in their use, including interactive, aimed at the manipulation of neutrosophic sets and functions. The structure of the entire framework has been carefully described by means of of Unified Modeling Language (UML for short), a modeling and specification description language very popular in Software Engineering. The underlying given structures as well as the most significant methods of each of the classes of which the the framework have been explained in detail in order to allow for further future refinements of both a theoretical and applicative nature.

We are confident that the necessity mentioned has been effectively tackled through the Python framework outlined in this paper. The complete source code for this framework has been released under the Open Source GNU General Public License version 3.0 (or GPL-3.0) and is freely accessible at the url github.com/giorgionordo/pythonNeutrosophicSets.

In particular, Section 2 describes the general structure of the framework, the dependency relationships among the various classes that comprise it as well as the reasons that suggested

the use of the Puython language and the decision to release the entire code produced under an Open Source license.

Section 3 introduces some useful notations for extending some data structures typical of Python in order to create a flexible substandard for describing neutrosophic sets and functions between them. In addition, some utility functions are briefly called that later will be invoked by some classes in the framework.

Section 4 contains a description of the properties and methods of the class `NSuniverse` used to represent the universe sets on which the neutrosophic sets will be defined.

Section 5 describes in great detail the properties and methods that make up the `NSset` class, the main class of the framework, used for the representation of neutrosophic sets and, in addition, numerous examples of practical use are also provided both in both traditional and interactive environments.

Section 6 is devoted to the description of the class `NSmapping` by which functions between two neutrosophic sets are represented. The properties and methods of this class are described in detail and illustrated with several practical examples.

Finally, some final remarks are made in Section 7, highlighting the strengths of the framework presented in this paper and inviting other researchers to continue, extend and improve the development of the code described here.

2. The framework **PYNS**

The PYthon Neutrosophic Sets framework (`PYNS` for short) described in the present paper consists of three classes designed to manage respectively universe sets (the `NSuniverse` class), neutrosophic sets (the `NSset` class) and functions between them (the `NSmapping` class. As is natural, the class `NSset` depends on (i.e. uses) the class `NSuniverse` class) while the `NSmapping` class uses the other two as described in the following UML diagram of classes.
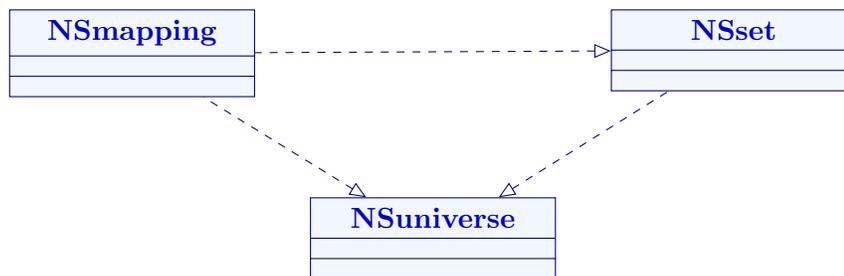
where the dashed arrow means "uses".

These three classes are respectively contained in the Python files `ns_universe.py`, `ns_set.py` and `ns_mapping.py` which are located in the package directory `pyns`. The same directory contains also the file `ns_util.py` where are defined some utility functions

external to the classes but employed by them. The structure of the package is described by the following diagram.

```
/pyns
│
├─── ns_universe.py ................(contains the NSuniverse class)
│
├─── ns_set.py ............................ (contains the NSset class)
│
├─── ns_mapping.py ...................(contains the NSmapping class)
│
└─── ns_util.py .................. (contains common utility functions)
```

The choice of programming language is a crucial aspect in the development of a scientific framework, since it determines the performance, flexibility and ease of use of the entire system. In the specific case of our framework, the choice to implement it using the Python language is based on several factors:

- Clear and expressive syntax: Python is known for its simple and readable syntax, which makes the code more intuitive to write and understand. This feature is especially relevant for a scientific framework, as it facilitates the creation and manipulation of complex data structures such as neutrosophic sets and mappings between them.
- Extensive standard library: Python offers an extensive standard library covering multiple scientific and mathematical domains. This allows developers to easily use existing functions and tools to implement complex algorithms and optimize the performance of the framework.
- Easy integration with other technologies: Python is known for its ability to integrate with other programming languages and external libraries. This is particularly useful in a scientific context, where it may be necessary to use specialized libraries or existing computational tools.
- Quick learning: Python is often considered one of the most accessible languages even for programming novices. Its relatively smooth learning curve allows students, researchers and less experienced developers to tackle the framework with greater ease, thus encouraging its dissemination and adoption in the scientific domain.

In summary, the choice of Python as the main language for the PYNS framework was crucial in making the entire system more accessible, flexible, and powerful. It allowed developers to focus more on scientific challenges and mathematics specifics, rather than the complexities of the programming language, thus accelerating the development and adoption of this important research tool.

Just as the Python language is released under an open source license approved by the Open Source Initiative (OSI), the framework described in this paper is also made available as

open code. This offers numerous benefits and incentives for both developers and the scientific community as a whole, including:

- Knowledge sharing: The release of the framework under an open source license promotes the sharing of knowledge and scientific discoveries. By allowing anyone to access the source code, developers and researchers can learn from others, build on others' work, and contribute improvements and new ideas.
- Collaboration: The open source license encourages collaboration among experts and researchers from from different academic institutions, organizations and countries. This synergy can lead to faster developments, new discoveries, and innovative solutions to complex problems.
- Transparency and verifiability: The availability of source code allows for greater transparency in the implementation of the framework. The scientific community can verify and validate the results obtained, increasing confidence towards the framework and the results obtained through it.
- Adaptability and customization: Users can adapt the framework to their own specifics and customize it to address unique problems. This flexibility results in a greater number of possible applications and uses of the framework in various scientific contexts.
- Cost reduction: Releasing the framework as open source eliminates the costs associated of purchasing licenses or copyrights. This allows academic institutions and organizations with limited resources to free access to advanced scientific analysis tools and uncertain data.
- Community growth: The adoption of the open source license attracts a community of developers, researchers and enthusiasts interested in the field of neutrosophic sets that can contribute to the evolution of the framework by providing feedback, reporting bugs, and participating in the development of new features.
- Continuity and longevity: The open source model can ensure greater longevity of the framework, as it is not dependent on a single developer or institution. The community can take care of the project over time, ensuring that it is always updated and supported, even if there are changes in the original organization.

In particular, our framework PYNS is available under GNU General Public License version 3.0 (or GPL-3.0), a generic software license developed by the Free Software Foundation (FSF) that provides users with a set of rights and freedoms to use, modify, and distribute the software covered by the license. More specifically, GPL-3.0 allows:

- Freedom of Use: the software may be used for any purpose, whether personal or commercial.

- Freedom to Study and Modify: you can analyze and study the source code of the software to understand how it works and make changes to it according to your own needs.

- Freedom of Distribution: you can distribute copies of modified or unmodified software to anyone, while complying with the requirements of GPL-3.0.

- Sharing of Changes: if you distribute modified software, you have an obligation to make available the source code of your changes as well.

- Compatibility with Derivative Works (copyleft): any derivative work based on software covered by GPL-3.0 must also be released under GPL-3.0 or a compatible license.

In conclusion, from the evaluation of all these aspects, it follows that releasing the `PYNS` framework under GPL-3.0 license represents a strategic choice that promotes innovation, collaboration and the dissemination of knowledge in the field of neutrosophic theory, thus promoting the advancement of scientific research in this area.

3. Conventions and utility functions

In the following we will make extensive use of Python's `dict` (dictionary) data structure both for the internal representation of neutrosophic sets and for the definition of functions between universe sets. To make it even easier and more streamlined to use such structures both in interactive mode as well as in writing client code based on such classes, it was chosen to also allow their representation as a string and in free format, i.e., leaving the user free to:

- indifferently use not only the usual symbol `:` (colon) but also alternatively the strings `->` (arrow) and `|->` (maps-to) as separators between keys and values

- indifferently use not only the usual symbol `,` (comma) but also `;` (semicolon) as separators of the value-key pairs

in any combination thereof, and we will refer to this type of representation by the name *extended dictionary*. In other words, while a classical Python dictionary has a form like:

$$\{key_1 : value_1, key_2 : value_2, \ldots key_n : value_n\} \,,$$

an extended dictionary can be expressed as strings of the type:

$$\texttt{"} key_1 \texttt{->} value_1 \texttt{,} key_2 \texttt{|->} value_2 \texttt{;} \ldots key_n \texttt{->} value_n \texttt{"} \,.$$

The already mentioned file `NS_util.py` contains some general utility functions that will be used repeatedly in the classes we will describe later. More specifically, these functions are:

- NSreplace(*text*, *sostituz*) which performs a series of substitutions on the string *text* by replacing each key in the *sostituz* dictionary with its corresponding value; in particular, if that value is the null string `""` the effect will be to remove all occurrences of the key,

- NSstringToTriplesList(*text*) which converts the string *text* containing a list of triples into the corresponding data structure by using the function `findall` contained in the module `re` (regular expression) and the function `literal_eval` contained in the module `ast` (Abstract Syntax Trees) which allows interpreting data expressions contained in a string,

- NSisExtDict(*obj*) that checks whether the object *obj* passed as parameter is a string representing an extended dictionary and returns the Boolean value `True` if it is,

- NSstringToDict(*text*) which converts the string *text* containing an extended dictionary in a real Python dictionary,

- NSsplitText(*text*, *max_length*) which returns the string *text* splitted into multiple lines of length not exceeding the value *max_length*.

The complete code for these functions is given in the following listing.

```python
from re import findall
from ast import literal_eval

def NSreplace(text, sostituz):
    for k in sostituz:
        text = text.replace(k, sostituz[k])
    return text

def NSstringToTriplesList(text):
    pattern = r'\[.*?\]|\(.*?\)'
    str_list = findall(pattern, text)
    tpl_list = [tuple(literal_eval(s)) for s in str_list]
    return tpl_list

def NSisExtDict(obj):
    result = False
    if type(obj) == str:
        result = (":" in obj) or ("->" in obj)

def NSstringToDict(text):
    sostituz = {"'": "", '"': "", "(": "", ")": "", "[": "", "]": "",
                "{": "", "}": "", " ": ",", ";": ",", ",,": ",",
                "|->": ":", "->": ':'}
    text = NSreplace(text, sostituz)
    listcouples = text.split(',')
    diz = dict()
    for couple in listcouples:
        key, value = couple.split(':')
        diz[key] = value
    return diz
```
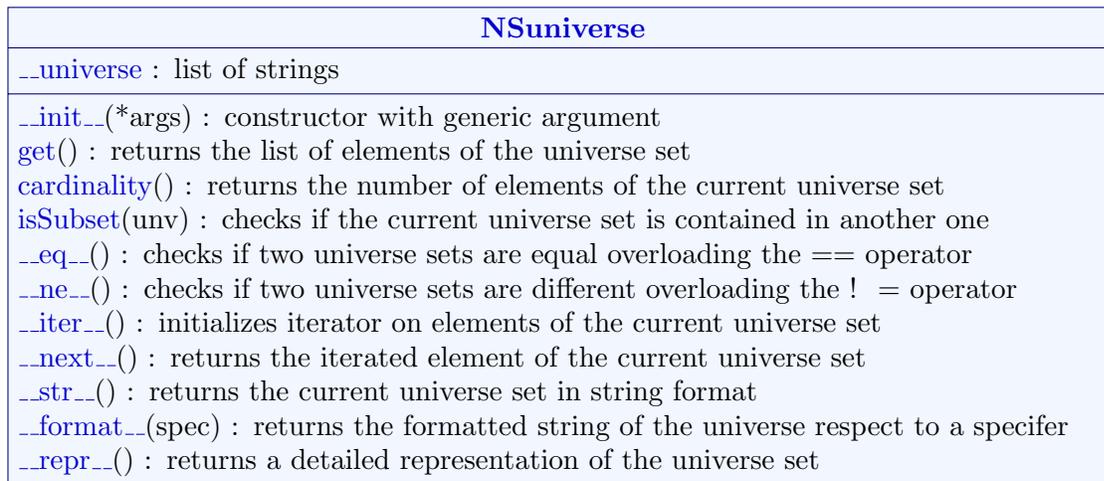
```
32 def NSsplitText(text, max_length):
33     words = text.split()
34     lines = []
35     current_line = ""
36     for word in words:
37         if len(current_line) + len(word) <= max_length:
38             current_line += word + " "
39         else:
40             lines.append(current_line.strip())
41             current_line = word + " "
42     lines.append(current_line.strip())
43     result = "\n".join(lines)
44     return result
```

4. The `NSuniverse` class

The universe set is the fundamental notion on which the definition of a neutrosophic set is founded on. We have chosen to represent it by means of a list of strings.

The corresponding class which implements such a notion is shortly described in the following UML class diagram.

NSuniverse
__universe : list of strings
__init__(*args) : constructor with generic argument get() : returns the list of elements of the universe set cardinality() : returns the number of elements of the current universe set isSubset(unv) : checks if the current universe set is contained in another one __eq__() : checks if two universe sets are equal overloading the == operator __ne__() : checks if two universe sets are different overloading the ! = operator __iter__() : initializes iterator on elements of the current universe set __next__() : returns the iterated element of the current universe set __str__() : returns the current universe set in string format __format__(spec) : returns the formatted string of the universe respect to a specifer __repr__() : returns a detailed representation of the universe set

In order to ensure maximum usability and versatility in the use of this class, the constructor method accepts string, lists, tuples, lists of elements of any length or another object `NSuniverse` and proceeds to transform them into strings and store them in a list. The basic steps of this method, expressed in pseudo-code, are described in the following algorithm.

Constructor method of the class `NSuniverse`

Function `__init__`(*args*):

Get the *length* of *args*

if *length* = 0 **then**

> Raise an Exception

else if *length* = 1 **then**

> **if** *args* **is** *a list, a tuple or an object of the class* **then**
>> Converts *args* appropriately and stores it in *universe*
>
> **else if** *args* **is** *a string* **then**
>> Removes parentheses, commas and semicolons from *args*, splits and gets a list of strings to store in *universe*
>
> **else if** *args* **is** *a set* **then**
>> Raise an Exception
>
> **else**
>> Converts *args* to string and creates a list with only this element to be stored in *universe*

else

> Converts *args* to a list of strings to be stored in *universe*

if *universe has repeated elements* **then**

> Raise an Exception

Stores *universe* in the property *__universe*

The corresponding Python code of the constructor method of the `NSuniverse` class is given below.

```python
1  from .ns_util import NSreplace
2
3  class NSuniverse:
4
5      def __init__(self, *args):
6          universe = list()
7          length = len(args)
8          if length == 0:
9              raise IndexError("the universe set must contain at least an
   element")
10         elif length == 1:
11             elem = args[0]
12             if type(elem) in [list, tuple]:
13                 universe = [str(e) for e in elem]
14             elif type(elem) == NSuniverse:
15                 universe = elem.get()
16             elif type(elem) == str:
17                 sostituz = { "{":"", "}":"", "[":"", "]":"", "(":"", ")":"",
18                              ",":" ", ";":" " }
19                 universe = NSreplace(elem, sostituz).split()
20             elif type(elem) == set:
21                 raise ValueError("type set is not suitable because the
   elements of the universe set must be assigned in a specific order")
22             else:
23                 universe = [str(elem)]
24         else:
```

```
25              for i in range(length):
26                  universe.append(str(args[i]))
27          univset = set(universe)
28          if len(universe) != len(univset):
29              raise ValueError("the universe set cannot contain repeated
     elements")
30          self.__universe = universe
```

The constructor method also intercepts potential error situations in the definition of universe sets such as attempting to define an empty set by calling it without any parameters or that of inserting repeated elements (which conflicts with the usual set definition) and in each of these cases raises an appropriate exception.

Let us observe that the flexibliity of the constructor method allows us to define a universe set using various formats such as lists, tuples, strings, or simple enumerations of elements without worrying about maintaining a rigid or uniform notation, which is particularly useful to facilitate usability in interactive use. For example, the universe set $\mathbb{U} = \{1, 2, 3, 4, 5\}$ can be defined as an object of the class `NSuniverse` in any of the following ways mutually equivalent:

- `U=NSuniverse([1,2,3,4,5])` as a list,
- `U=NSuniverse((1,2,3,4,5))` as a tuple,
- `U=NSuniverse("1,2,3,4,5")` as a string of elements comma separated,
- `U=NSuniverse("1;2;3;4;5")` as a string of elements separated by semicolon,
- `U=NSuniverse("1 2 3 4 5")` as a string of elements separated by spaces,
- `U=NSuniverse("1,2 3 4;5")` as a string of elements separated in various ways,
- `U=NSuniverse("{1,2,3,4,5}")` as a string representing a set,
- `U=NSuniverse("[1,2,3,4,5]")` as a string representing a list,
- `U=NSuniverse("(1,2,3,4,5)")` as a string representing a tuple,
- `U=NSuniverse("(1;2;3;4;5)")` using semicolon as separator,
- `U=NSuniverse(1,2,3,4,5)` as a listing of numerical values only,
- `U=NSuniverse("1",2,"3",4,"5")` in a mixed form,

as well as in different combinations of them.

However, it is not allowed to define a universe set by means of the `set` type of the Python language (i.e., expressions such as `NSuniverse({1,2,3,4,5})` are not accepted) since it is an unordered data collection and for a precise design choice the elements must be listed in a specific order, feature this will prove valuable in simplifying and making consistent definitions of both neutrosophic sets and mappings between them.

The class `NSuniverse` is equipped with very few basic methods, that is `get()` which returns the list of strings corresponding to the instance of the universe set exactly as it is stored internally in the class and `cardinality()` which returns the number of elements present in the object instantiated by the class.

```
1    def get(self):
2        return self.__universe

4    def cardinality(self):
5        return len(self.__universe)
```

The method `isSubset()` checks whether the current universe set is contained in a second universe set passed as parameter and returns the Boolean value `True` in the positive case.

```
1    def isSubset(self, unv):
2        setself = set(self.get())
3        setunv = set(unv.get())
4        result = setself.issubset(setunv)
5        return result
```

To facilitate the comparison of two objects of type `NSuniverse`, the equality `==` and diversity `!=` operators have been overloaded using their corresponding special methods.

```
1    def __eq__(self, unv):
2        equal = (self.get() == unv.get())
3        return equal

5    def __ne__(self, unv):
6        different = not (self == unv)
7        return different
```

In order to be able to easily print on the screen objects of type `NSuniverse` in text format and to provide a complete representation of them, the special methods `__str__()` and `__repr__()` were defined as follows by using the overloading.

```
1    def __str__(self):
2        list_string_elements = [str(e) for e in self.__universe]
3        s = "{ " + ", ".join(list_string_elements) + " }"
4        return s

6    def __repr__(self):
7        return f"Universe set: {str(self)}"
```

As an example, we show how the methods and operators described above can be used not only in a client code, but also in the interactive mode by means of the Python console:

```
>>> from pyns.ns_universe import NSuniverse
>>> U = NSuniverse("1", 2, 3, "4")
>>> print(U)
{ 1, 2, 3, 4 }
>>> print( U != NSuniverse([1,3,5]))
True
>>> V = NSuniverse(" ( a b c , d ; e )")
>>> print(V.cardinality())
5
>>> print(V.get())
['a', 'b', 'c', 'd', 'e']
>>> print(V)
{ a, b, c, d, e }
```

Since in the following we will also need to make formatted prints of objects of the universe set type according to a certain format specifier, it is also necessary to redefine by overloading the special method `__format__()`.

```python
def __format__(self, spec):
    unvstr = str(self)
    result = f"{unvstr:{spec}}"
    return result
```

Finally, to simplify the code of other classes devoted to neutrosophic sets and mappings, it is useful to establish an iterator over the objects of the `NSuniverse` class. This iterator should sequentially provide all and only the elements within a given universe set. This is achieved by introducing a new property, $self.\_\_i$, to serve as the internal index for the current element and redefining by overloading the special methods `__iter__()` and `__next__()` which are respectively intended to initialize the index of the iterator and yield the element associated with the current index.

```python
def __iter__(self):
    self.__i = 0
    return self

def __next__(self):
    if self.__i < len(self.__universe):
        elem = self.__universe[self.__i]
        self.__i +=1
        return elem
    raise StopIteration
```

Thanks to the introduction of the iterator on the class `NSuniverse` it will be, for example, possible to handle loops directly on objects of type universe set, exactly as happens with other standard Python types such as lists and tuples. This approach will contribute to make the syntax of our code leaner and more understandable, as highlighted in the following example.

```python
from pyns.ns_universe import NSuniverse

U = NSuniverse(" ( a b c , d ; e )")
for i, u in enumerate(U):
    print(f"- the {i}-th element is {u}")
```

which produces output of the type:

```
- the 0-th element is a
- the 1-th element is b
- the 2-th element is c
- the 3-th element is d
- the 4-th element is e
```

5. The `NSset` class

The representation of neutrosophic sets on a given universe set is done by means of the `NSset` class which, obviously, uses the `NSuniverse` class.

The original definition of neutrosophic set, given in 1999 by Smarandache [27], refers to the interval $]0^-, 1^+[$ of the nonstandard real numbers and although it is consistent from a philosophical point of view, unfortunately, it is not suitable to be used for approaching real-world problems. For such a reason, in 2010, the same author, jointly with Wang, Zhang and Sunderraman [30], also introduced the notion of single valued neutrosophic set which, referring instead to the unit interval $[0,1]$ of the usual set of real numbers $\mathbb{R}$, can be usefully used in scientific and engineering applications. In the following we will refer exclusively to single valued neutrosophic sets.

Definition 5.1. [30] Let $\mathbb{U}$ be an universe set and $A \subseteq \mathbb{U}$, a *single valued neutrosophic set* over $\mathbb{U}$ (*SVN-set* for short), denoted by $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$, is a set of the form:

$$\widetilde{A} = \{(u, \mu_A(u), \sigma_A(u), \omega_A(u)) : u \in \mathbb{U}\}$$

where $\mu_A : \mathbb{U} \to I$, $\sigma_A : \mathbb{U} \to I$ and $\omega_A : \mathbb{U} \to I$ are the *membership function*, the *indeterminacy function* and the *non-membership function* of A, respectively and $I = [0,1]$ be the unit interval of the real numbers. For every $u \in \mathbb{U}$, $\mu_A(u)$, $\sigma_A(u)$ and $\omega_A(u)$ are said the *degree of membership*, the *degree of indeterminacy* and the *degree of non-membership* of u, respectively.

Definition 5.2. [27,30] Let $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$ be two SVN-sets over the universe set $\mathbb{U}$, we say that $\widetilde{A}$ is a *neutrosophic subset* (or simply a subset) of $\widetilde{B}$ and we write $\widetilde{A} \subseteq \widetilde{B}$ if, for every $u \in \mathbb{U}$, it results $\mu_A(u) \leq \mu_B(u)$, $\sigma_A(u) \leq \sigma_B(u)$ and $\omega_A(u) \geq \omega_B(u)$. We also say that $\widetilde{A}$ is contained in $\widetilde{B}$ or that $\widetilde{B}$ contains $\widetilde{A}$ and we write $\widetilde{B} \supseteq \widetilde{A}$ to denote that $\widetilde{B}$ is a *neutrosophic superset* of $\widetilde{A}$.

Definition 5.3. [27,30] Let $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$ be two SVN-sets over the universe set $\mathbb{U}$. We say that $\widetilde{A}$ is a *neutrosophically equal* (or simply equal) to $\widetilde{B}$ and we write $\widetilde{A} \equiv \widetilde{B}$ if $\widetilde{A} \subseteq \widetilde{B}$ and $\widetilde{B} \subseteq \widetilde{A}$.

Notation 1. *Let $\mathbb{U}$ be a set, $I = [0,1]$ the unit interval of the real numbers, for every $r \in I$, with $\underline{r}$ we denote the constant mapping $\underline{r} : \mathbb{U} \to I$ defined by $\underline{r}(u) = r$, for every $u \in \mathbb{U}$.*

Definition 5.4. [30] The SVN-set $\langle \mathbb{U}, \underline{0}, \underline{0}, \underline{1} \rangle$ is said to be the *neutrosophic empty set* over $\mathbb{U}$ and it is denoted by $\widetilde{\emptyset}$, or more precisely by $\widetilde{\emptyset}_{\mathbb{U}}$ in case it is necessary to specify the corresponding universe set.

Definition 5.5. [30] The SVN-set $\langle \mathbb{U}, \underline{1}, \underline{1}, \underline{0} \rangle$ is said to be the *neutrosophic absolute set* over $\mathbb{U}$ and it is denoted by $\widetilde{\mathbb{U}}$.

In our class, the data structure used to represent a SVN-set is a dictionary (also called associative array) which uses the elements of the universe set as keys and associates them with a list of three floating-point numbers corresponding to the degrees of membership, indeterminacy and non-membership respectively. This dictionary, referred as the `__neutrosophicset` property, is stored in conjunction with the universe set, referred as the `__universe` property, to which it is inseparably linked. Indeed, it is no coincidence that the class `NSuniverse` does not provide any method for allowing modifications (such as insertions or removals) of the elements of an object of type universe set since such operations could disrupt the consistency of the SVN-sets defined on it. The `NSset` class is described by the following UML diagram.

NSset
__universe : object of the class NSuniverse
__neutrosophicset : dictionary of lists
__init__(*args) : constructor with generic arguments
__setDegree(u,i,r) : assigns r to the i-th degree of element u
setMembership(u, mu) : assigns the membership degree to u
setIndeterminacy(u, sigma) : assigns the indeterminacy degree to u
setNonMembership(u, omega) : assigns the non-membership degree to u
setElement(u, triple) : assigns simultaneously the three degrees to u
getUniverse() : returns the universe of the SVN-set
get() : returns the dictionary containing the degrees of each element
getElement(u) : returns the triple the degrees of the element u
__getDegree(u,i) : returns the i-th degree of element u
getMembership(u) : returns the membership degree of u
getIndeterminacy(u) : returns the indeterminacy degree of u
getNonMembership(u) : returns the non-membership degree of u
setEmpty() : makes equal to the neutrosophic empty set
setAbsolute() : makes equal to the absolute SVN-set
cardinality() : returns the number of elements of the SVN-set
isNSsubset(nset) : checks if is a neutrosophic subset of another one
isNSsuperset(nset) : checks if is a neutrosophic superset of another one
NSunion(nset) : returns the neutrosophic union with another one
NSintersection(nset) : returns the neutrosophic intersection with another one
isNSdisjoint(nset) : checks if is neutrosophically disjoint with another one
NScomplement() : returns the neutrosophic complement
NSdifference(nset) : returns the neutrosophic difference with another one
__eq__() : checks if two SVN-sets are equal overloading the == operator
__ne__() : checks if two SVN-sets are different overloading the ! = operator
__add__(nset) : neutrosophic union overloading the + operator
__and__(nset) : neutrosophic intersection overloading the & operator
__invert__() : neutrosophic complement overloading the ∼ operator
__sub__(nset) : neutrosophic difference overloading the − operator
__le__(nset) : neutrosophic subset overloading the <= operator
__ge__(nset) : neutrosophic superset overloading the >= operator
__str__() : returns the SVN-set in string format
__format__(spec) : returns the formatted string of the SVN-set respect to a specifer
__repr__() : returns a detailed representation of the SVN-set

The constructor method of this class accepts one or two parameters and allows us to define a SVN-set in several different ways:

- in the form NSset (*universe set*) assigning to it as its only parameter the universe set (expressed as a list, tuple, or string) over which it is defined and thus creating an empty SVN-set $\widetilde{\emptyset}$,

- in the form NSset (*universe set*, *values*) passing two parameters, the first of which is a universe set and the second an enumeration (expressed as a list, tuple, or string) of triples of real values representing the degree of membership, indeterminacy and non-membership of all the elements of the universe set, or

- in the form NSset (*neutrosophic set*) by copying another object of the type NSset.

The basic steps of this method, expressed in pseudo-code, are described in the following algorithm.

Constructor method of the class NSset

Function __init__(*args*):

    Create a dictionary *neutrosophicset*

    Get the *length* of *args*

    if *length* = 1 **then**

        if *args* **is** *a list, a tuple, a string or an object of the class* NSuniverse **then**

            Create *universe* from *args* and set *neutrosophicset* empty

        else if *args* **is** *an object of the class* NSset **then**

            Get *universe* and *neutrosophicset* from *args*

        else

            Raise an Exception

    else if *length* = 2 **then**

        Use the same constructor with the first parameter of *args* to obtain an object of type NSset from which to derive *universe* and set the list *values* equal to the second parameter of *args*

        if *values* **is** *a list or a tuple* **then**

            if *length of values is different from the length of universe* **then**

                Raise an Exception

            Assigns to each element of *universe* the values of the corresponding triple of *values*

        else if *args* **is** *a string* **then**

            Converts *args* to a list of triples and uses the same constructor with *universe* and such a list to obtain an object of type NSset from which to take *neutrosophicset*

        else

            Raise an Exception

    else

        Raise an Exception

    Stores *universe* and *neutrosophicset* in the properties __*universe* and __*neutrosophicset* respectively

---

Note how every possible error condition – such as, for example, attempting to pass a variable of type other than `NSuniverse` or a list of values not consisting of triples of real numbers included in the $[0, 1]$ interval or, again, using it with fewer than one or more than two parameters – is intercepted in the code and reported to the client by raising an appropriate exception.

The Python code corresponding to the constructor method of the `NS_set` class is given below.

```python
from .ns_universe import NSuniverse
from .ns_util import NSreplace, NSstringtoTriplesList, NSsplitText

class NSset:

    degreename = ["membership", "indeterminacy", "non-membership"]
    reprmaxlength = 64

    def __init__(self, *args):
        neutrosophicset = dict()
        length = len(args)
        if length == 1:
            element = args[0]
            if type(element) in [list, tuple, str, NSuniverse]:
                universe = NSuniverse(element)
                for e in universe.get():
                    neutrosophicset[e] = [0,0,1]
            elif type(element) == NSset:
                universe = element.getUniverse()
                for e in universe:
                    neutrosophicset[e] = element.getElement(e)
            else:
                raise ValueError("value not compatible with the type universe
    set")
        elif length == 2:
            nset = NSset(args[0])
            universe = nset.getUniverse()
            values = args[1]
            if type(values) in [list ,tuple]:
                if len(values) != len(universe):
                    raise IndexError("the number of value triples does not
    correspond with the number of elements")
                for i in range(len(universe)):
                    elem = universe[i]
                    t = values[i]
                    if type(t) not in [tuple,list] or len(t) !=3:
                        raise IndexError("the second parameter of the
    constructor method must contain only triple")
                    t = [float(t[j]) for j in range(3)]
                    for j in range(3):
                        if not 0 <= t[j] <= 1:
                            raise ValueError(f"incompatible {self.degreename[j
    ]} degree value")
                    neutrosophicset[elem] = t
            elif type(values) == str:
                tpl_list = NSstringtoTriplesList(values)
                nset = NSset(universe, tpl_list)
```

```
44              neutrosophicset = nset.get()
45          else:
46              raise ValueError("the second parameter of the constructor
    method must contain a list of triples of real numbers")
47          else:
48              raise IndexError("the number of parameters do not match those of
    the constructor method")
49          self.__universe = NSuniverse(universe)
50          self.__neutrosophicset = neutrosophicset
```

Let us observe that to enable the parsing of strings containing triples with the values of membership degrees, indeterminacy and non-membership, variously expressed, were used the function `NSstringToTriplesList` contained in the utility file `NS_util.py`.

Similar to what we have seen previously for objects of type universe set, the constructor method of the class `NSset` also allows for a multiplicity of expressions thanks to which we can define neutrosophic sets with a direct and informal notation by means of lists, tuples or strings of elements separated indifferently by commas or semicolons.

For example, the SVN-set $\langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ over the universe set $\mathbb{U} = \{a, b, c\}$ defined by:

$$\left\langle \frac{a}{(0.5, 0.3, 0.2)}, \frac{b}{(0.6, 0.2, 0.3)}, \frac{c}{(0.4, 0.2, 0.7)} \right\rangle$$

can be defined as an object of the class `NSset` in any of the following ways mutually equivalent:

- `A=NSset("a,b,c",[(0.5,0.3,0.2),(0.6,0.2,0.3),(0.4,0.2,0.7)])` as a list of tuples,

- `A=NSset("a,b,c",[[0.5,0.3,0.2],[0.6,0.2,0.3],[0.4,0.2,0.7]])` as a list of lists,

- `A=NSset("a,b,c",[[0.5,0.3,0.2],(0.6,0.2,0.3),[0.4,0.2,0.7]])` as a mixed list of lists and tuples,

- `A=NSset("a,b,c",([0.5,0.3,0.2],[0.6,0.2,0.3],[0.4,0.2,0.7]))` as a tuple of lists,

- `A=NSset("a,b,c",((0.5,0.3,0.2),(0.6,0.2,0.3),(0.4,0.2,0.7)))` as a tuple of tuples,

- `A=NSset("a,b,c",((0.5,0.3,0.2),[0.6,0.2,0.3],(0.4,0.2,0.7)))` as a mixed tuple of tuples and lists,

- `A=NSset("a,b,c","[0.5,0.3,0.2],(0.6,0.2,0.3);[0.4,0.2,0.7]")` as a string containing lists and tuples

where the universe set can also be expressed in any equivalent form as `U=NSuniverse("{a,b,c}")`, `U=NSuniverse("[a,b;c]")` or `U=NSuniverse("(a;b,c)")` so that it can be used later in the definition of the SVN-set in the form:

- `A=NSset(U,"[0.5,0.3,0.2],(0.6,0.2,0.3);[0.4,0.2,0.7]")`

A SVN-set $\langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ already defined over a universe set $\mathbb{U}$ can be subsequently modified by assigning to each of its generic elements $u \in \mathbb{U}$ its degrees of membership $mu = \mu_A(u)$, indeterminacy $sigma = \sigma_A(u)$ and non-membership $omega = \omega_A(u))$ separately through the methods:

- setMembership(`self`, u, mu)
- setIndeterminacy(`self`, u, $sigma$), and
- setNonMembership(`self`, u, $omega$)

or by assigning in one shot the entire $triple = (\mu_A(u), \sigma_A(u), \omega_A(u))$ of values using the method:

- setElement(`self`, u, $triple$)

all of which are based on the private method __setDegree(`self`, u, i, v) that assigns the value v to the i-th degree (for $i = 0, 1, 2$, which correspond in the order to membership, indeterminacy and non-membership degree) of a given element u of the current SVN-set. Obviously, in the latter method we take into account the fact that the element u must belong to the corresponding universe set and that the membership degrees must be real values included in the unit interval I and if not, appropriate exceptions will be raised.

```python
def __setDegree(self, u, i, r):
    u = str(u)
    if u not in self.getUniverse():
        raise IndexError('non-existent element')
    r = float(r)
    if not (0 <= r <= 1):
        raise ValueError(f"incompatible {self.degreename[i]} degree value"
)
    self.__neutrosophicset[u][i] = r

def setMembership(self, u, mu):
    self.__setDegree(u, 0, mu)

def setIndeterminacy(self, u, sigma):
    self.__setDegree(u, 1, sigma)

def setNonMembership(self, u, omega):
    self.__setDegree(u, 2, omega)

def setElement(self, u, triple):
    if type(triple) == str:
        sostituz = { "(":"", ")":"", ",":" ", ";":" " }
        triple = NSreplace(triple, sostituz).split()
    else:
        triple = list(triple)
    if len(triple) != 3:
        raise ValueError('error in the number of parameters passed')
    triple = [float(e) for e in triple]
    for i in range(3):
        self.__setDegree(u, i, triple[i])
```

Three basic methods called `getUniverse()`, `get()` and `getElement(u)` respectively return to us the universe set of a given SVN-set as a string list of its elements, the neutrosophic set itself as a dictionary having for keys the elements of the universe and for values the triples of the degrees of membership, indeterminacy and non-membership as well as the triple of the degrees of a given element $u \in \mathbb{U}$.

```python
def getUniverse(self):
    return self.__universe.get()

def get(self):
    return self.__neutrosophicset

def getElement(self, u):
    u = str(u)
    if u not in self.getUniverse():
        raise IndexError('non-existent element')
    return self.__neutrosophicset[u]
```

Given a SVN-set $\langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ over a universe set $\mathbb{U}$, The values of the degrees of membership $\mu_A(u)$, indeterminacy $\sigma_A(u)$ and non-membership $\omega_A(u)$ of a generic element $u \in \mathbb{U}$ can be obtained by the methods:

- `getMembership(`u`)`,
- `getIndeterminacy(`u`)`, and
- `getNonMembership(`u`)`

which are all based on the private method `__getDegree(`$self, u, i$`)` that returns the value of the i-th degree (for $i = 0, 1, 2$, which correspond in the order to membership, indeterminacy and non-membership degree) of a given element u of the current SVN-set. Let us note that in the latter method, we consider the prerequisite that the element u must belong to the corresponding universe set and if this condition is not satisfied a suitable exception will be raised.

```python
def __getDegree(self, u, i):
    u = str(u)
    if u not in self.getUniverse():
        raise IndexError('non-existent element')
    return self.__neutrosophicset[u][i]

def getMembership(self, u):
    return self.__getDegree(u, 0)

def getIndeterminacy(self, u):
    return self.__getDegree(u, 1)

def getNonMembership(self, u):
    return self.__getDegree(u, 2)
```

In case we need the empty neutrosophic set $\widetilde{\emptyset}$ or the neutrosophic absolute set $\widetilde{\mathbb{U}}$ over an universe set $\mathbb{U}$ we can refer to the methods `setEmpty()` and `setAbsolute()`, respectively.

```python
def setEmpty(self):
    for e in self.__universe.get():
        self.__neutrosophicset[e] = [0, 0, 1]

def setAbsolute(self):
    for e in self.__universe.get():
        self.__neutrosophicset[e] = [1, 1, 0]
```

The on-screen printing in text format of objects of type `NSset` as well as their complete representation is achieved by overloading the special methods `__str__` and `__repr__` as follows.

```python
def __str__(self, tabularFormat=False):
    if tabularFormat == True:
        (dashes, elemwidth, valwidth) = ("-"*64, 10, 14)
        s = "\n                | membership  |  indeterminacy | non-
membership |\n" + dashes + "\n"
        for e in self.getUniverse():
            (mu, sigma, omega) = self.getElement(e)
            s += f" {str(e):{elemwidth}} | {mu:{valwidth}} | {sigma:{
valwidth}} | {omega:{valwidth}} |\n"
        s += dashes + "\n"
    else:
        elems = []
        for e in self.getUniverse():
            (mu, sigma, omega) = self.getElement(e)
            elems.append(f"{e}/({mu},{sigma},{omega})")
        s = "< " + ", ".join(elems) + " >"
        s = NSsplitText(s, self.reprmaxlength)
    return s

def __repr__(self):
    return f"Neutrosophic set: {str(self)}"
```

In particular, the special method `__str__` allow us to print on the screen a SVN-set in both the simplified representation (which is the default option) and in the clearer and more extensive tabular representation.

Furthermore, in order to be able to choose to print a SVN-set in the simplified representation or in the tabular one even in interactive use or writing client code, it was chosen to redefine the special method `__format__` so that it recognizes the new custom format specifier `t` that corresponds to printing in tabular format objects of type `NSset`.

```python
def __format__(self, spec):
    if spec == "t":
        result = self.__str__(tabularFormat=True)
    else:
        result = self.__str__(tabularFormat=False)
    return result
```

Thanks to the redefinition by overloading of the special methods `__eq__` and `__ne__` we can use the operators of equality `==` and diversity `!=` directly to objects of type `NSset`.

```python
def __eq__(self, nset):
    if self.getUniverse() != nset.getUniverse():
        raise ValueError("the two neutrosophic sets cannot be defined on
different universe sets")
    equal = self.isNSsubset(nset) and nset.isNSsubset(self)
    return equal

def __ne__(self, nset):
    if self.getUniverse() != nset.getUniverse():
        raise ValueError("the two neutrosophic sets cannot be defined on
different universe sets")
    different = not (self == nset)
    return different
```

In order to better illustrate how the above methods are used, let us consider the following example of code executed interactively in the Python console.

```python
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> U = NSuniverse("a,b,c")
>>> A = NSset(U)
>>> A.setElement('a', (0.8,0.2,0.1))
>>> A.setElement('c', (0.3,0.2,0.4))
>>> A.getMembership('a')
0.8
>>> A.getNonMembership('c')
0.4
>>> print(A.getElement(c))
[0.3, 0.2, 0.4]
>>> A.setIndeterminacy('b',0.9)
>>> print(A)
< a/(0.8,0.2,0.1), b/(0.0,0.9,1.0), c/(0.3,0.2,0.4) >
>>> print(f"{A:t}")
           |   membership   |  indeterminacy | non-membership |
------------------------------------------------------------------
 a         |            0.8 |            0.2 |            0.1 |
 b         |              0 |            0.9 |              1 |
 c         |            0.3 |            0.2 |            0.4 |
------------------------------------------------------------------
>>> A.setAbsolute()
>>> print(A)
< a/(1,1,0), b/(1,1,0), c/(1,1,0) >
```

From the example above, one might assume that creating universe sets and SVN-sets requires manual definition. However, the open structure of our framework actually enables us to define objects of type `NSuniverse` and `NSset` dynamically within the code, commonly referred to as defining them 'on the fly'. This dynamic approach is especially advantageous when dealing with SVN-sets of considerable cardinality, as illustrated in the following Python code.

```python
from pyns.ns_universe import NSuniverse
from pyns.ns_set import NSset
from random import random
```

```
 5 lst = [(i,j) for i in range(1,6) for j in range(1,4)]

 7 U = NSuniverse(lst)
 8 A = NSset(U)

10 for u in U.get():
11     triple = [round(random(), 2) for k in range(3)]
12     A.setElement(u, triple)

14 print(f"The following SVNS-set has cardinality {A.cardinality()}: {A:t}")
```

which produces output of the type:

```
The following SVNS-set has cardinality 15:
              |   membership   | indeterminacy | non-membership |
         ---------------------------------------------------------
  (1, 1)      |           0.55 |          0.1  |           0.5  |
  (1, 2)      |           0.39 |          0.92 |           0.09 |
  (1, 3)      |           0.33 |          0.29 |           0.25 |
  (2, 1)      |           0.16 |          0.9  |           0.43 |
  (2, 2)      |           0.71 |          0.52 |           0.33 |
  (2, 3)      |           0.65 |          0.38 |           0.04 |
  (3, 1)      |           0.95 |          0.14 |           0.94 |
  (3, 2)      |           0.74 |          0.02 |           0.01 |
  (3, 3)      |           0.77 |          0.63 |           0.19 |
  (4, 1)      |           0.18 |          0.75 |           0.15 |
  (4, 2)      |           0.49 |          0.92 |           0.75 |
  (4, 3)      |           0.34 |          0.17 |           0.88 |
  (5, 1)      |           0.88 |          0.6  |           0.83 |
  (5, 2)      |           0.5  |          0.56 |           0.8  |
  (5, 3)      |           0.38 |          0.47 |           0.2  |
         ---------------------------------------------------------
```

To verify that a SVN-set expressed as an object of type `NSset` is neutrosophically contained in another SVN-set, we may resort to the method `isNSsubset(`*nset*`)` which, similarly to the built-in `issubset()` method available for objects of type `set`, returns the Boolean value `True` if the current SVN-set is neutrosophically contained in the second SVN-set *nset* passed as a parameter or the value `False` otherwise.

```
 1     def isNSsubset(self, nset):
 2         if self.getUniverse() != nset.getUniverse():
 3             raise ValueError("the two neutrosophic sets cannot be defined on
         different universe sets")
 4         if self.getUniverse() != nset.getUniverse():
 5             return False
 6         else:
 7             result = True
 8             for e in self.getUniverse():
 9                 (muA, sigmaA, omegaA) = self.getElement(e)
10                 (muB, sigmaB, omegaB) = nset.getElement(e)
11                 if (muA > muB) or (sigmaA > sigmaB) or (omegaA < omegaB):
12                     result = False
13                     break
14             return result
```

Nordo G., Jafari S., Mehmood A., Basumatary B., A Python Framework for Neutrosophic Sets and Mappings

Based on `isNSsubset`, it is then immediate to define the method `isNSsuperset`(*nset*) (analogous to the built-in `issuperset` method) which returns `True` if the current SVN-set neutrosophically contains the second SVN-set *nset* passed as a parameter or `False` otherwise.

```
def isNSsuperset(self, nset):
    if self.getUniverse() != nset.getUniverse():
        raise ValueError("the two neutrosophic sets cannot be defined on
different universe sets")
    return nset.isNSsubset(self)
```

In both cases, it is preliminarily verified that the two SVN-sets are defined on the same universe set and if not an appropriate exception is raised.

The following code executed in interactive mode in the Python console illustrates the use of the methods just described.

```
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> A = NSset("a, b, c", "(0.3,0,0.5), (0.7,0.2,0.2), (0.1,0.5,0.4)")
>>> print(A)
< a/(0.3,0.0,0.5), b/(0.7,0.2,0.2), c/(0.1,0.5,0.4) >
>>> B = NSset("a, b, c", "(0.4,0.2,0.3), (0.8,0.3,0.1), (0.2,0.5,0.2)")
>>> print(B)
< a/(0.4,0.2,0.3), b/(0.8,0.3,0.1), c/(0.2,0.5,0.2) >
>>> print(A.isNSsubset(B))
True
>>> print(A.isNSsuperset(B))
False
```

Definition 5.6. [19] The *neutrosophic union* of two SVN-sets $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$, denoted by $\widetilde{A} \uplus \widetilde{B}$, is the neutrosophic set defined by $\langle \mathbb{U}, \mu_A \vee \mu_B, \sigma_A \vee \sigma_B, \omega_A \wedge \omega_B \rangle$.

Definition 5.7. [19] The *neutrosophic intersection* of two SVN-sets $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$, denoted by $\widetilde{A} \sqcap \widetilde{B}$, is the neutrosophic set defined by $\langle \mathbb{U}, \mu_A \wedge \mu_B, \sigma_A \wedge \sigma_B, \omega_A \vee \omega_B \rangle$.

Definition 5.8. [30] Let $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$ be two SVN-sets over $\mathbb{U}$. We say that $\widetilde{A}$ and $\widetilde{B}$ are *neutrosophically disjoint* if $\widetilde{A} \sqcap \widetilde{B} \equiv \widetilde{\emptyset}$. On the contrary, if $\widetilde{A} \sqcap \widetilde{B} \not\equiv \widetilde{\emptyset}$ we say that $\widetilde{A}$ *neutrosophically meets* $\widetilde{B}$ (or that $\widetilde{A}$ and $\widetilde{B}$ neutrosophically meet each other).

Within our `NSset` class, the neutrosophic union and neutrosophic intersection were implemented through the methods `NSunion()` and `NSintersection()`, respectively. These methods mirror the built-in Python methods `union` e `intersection` for objects of type `set` and they are both based on the private method `__NSoperation`(`self`, *nset*, *fm*, *fs*, *fo*)

corresponding to a generic operation and that returns the neutrosophic set obtained from the current SVN-set and the second SVN-set *nset* by applying the three functions *fm*, *fs*, *fo* passed as parameters to their membership, indeterminacy and non-membership degrees respectively.

```python
def __NSoperation(self, nset, fm, fs, fo):
    if self.getUniverse() != nset.getUniverse():
        raise ValueError("the two neutrosophic sets cannot be defined on
different universe sets")
    if callable(fm) == False or callable(fs) == False or callable(fo) ==
False:
        raise  ValueError("the last three parameters must be functions")
    C = NSset(self.__universe)
    for e in self.getUniverse():
        (muA, sigmaA, omegaA) = self.getElement(e)
        (muB, sigmaB, omegaB) = nset.getElement(e)
        triple = [fm(muA, muB), fs(sigmaA, sigmaB), fo(omegaA, omegaB)]
        C.setElement(e, triple)
    return C

def NSunion(self, nset):
    C = self.__NSoperation(nset, max, max, min)
    return C

def NSintersection(self, nset):
    C = self.__NSoperation(nset, min, min, max)
    return C
```

We illustrate the above methods with an example of code executed interactively in the Python console.

```python
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> U = NSuniverse("a, b, c")
>>> A = NSset(U,"(0.3,0,0.5), (0.7,0.2,0.2), (0.1,0.5,0.4)")
>>> print(A)
< a/(0.3,0.0,0.5), b/(0.7,0.2,0.2), c/(0.1,0.5,0.4) >
>>> B = NSset(U,"(0.4,0.2,0.3), (0.8,0.3,0.1), (0.2,0.5,0.2)")
>>> print(B)
a/(0.4,0.2,0.3), b/(0.8,0.3,0.1), c/(0.2,0.5,0.2) >
>>> C = A.NSunion(B)
>>> print(C)
< a/(0.4,0.2,0.3), b/(0.8,0.3,0.1), c/(0.2,0.5,0.2) >
>>> D = A.NSintersection(B)
>>> print(D)
< a/(0.3,0.0,0.5), b/(0.7,0.2,0.2), c/(0.1,0.5,0.4) >
```

The method `isNSdisjoint(nset)` returns the Boolean value `True` if the current SVN-set is neutrosophically disjoint from the second SVN-set passed as parameter or the value `False` otherwise.

```python
def isNSdisjoint(self, nset):
    nsempty = NSset(self.__universe)
    disjoint = self.NSintersection(nset) == nsempty
    return disjoint
```

An example of the use of this method is provided in the following code executed in the console Python interactive.

```
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> A = NSset("a, b, c", "(0.3,0,0.5), (0.7,0,1), (0,0.5,1)")
>>> B = NSset("a, b, c", "(0,0.8,1), (0.1,0.2), (0.1,0.0,0.4)")
>>> print(A.isNSdisjoint(B))
True
```

Definition 5.9. [27, 30] Let $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ be a SVN-set over the universe set $\mathbb{U}$, the *neutrosophic complement* (or, simply, the complement) of $\widetilde{A}$, denoted by $\widetilde{A}^{\complement}$, is the SVN-set $\widetilde{A}^{\complement} = \langle \mathbb{U}, \omega_A, \underline{1} - \sigma_A, \mu_A \rangle$ that is $\widetilde{A}^{\complement} = \{(u, \omega_A(u), 1 - \sigma_A(u), \mu_A(u)) : u \in \mathbb{U}\}$.

Although the neutrosophic difference of two SVN-sets $\widetilde{A}$ and $\widetilde{B}$ can be defined (in analogy with ordinary sets) as the neutrosophic intersection of the first set with the neutrosophic complement of the second set, that is, as $\widetilde{A} \cap \widetilde{B}^{\complement}$, for our purposes it is preferable to provide an explicit and operational definition.

Definition 5.10. The *neutrosophic difference* of two SVN-sets $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ and $\widetilde{B} = \langle \mathbb{U}, \mu_B, \sigma_B, \omega_B \rangle$, denoted by $\widetilde{A} \setminus\!\setminus \widetilde{B}$, is the neutrosophic set defined by $\langle \mathbb{U}, \mu_A \wedge \omega_B, \sigma_A \wedge (\underline{1} - \sigma_B), \omega_A \vee \mu_B \rangle$.

The latter operations have also been implemented in the class `NSset` through the methods `NScomplement()` and `NSdifference()`.

```
1      def NScomplement(self):
2          C = NSset(self.__universe)
3          for e in self.getUniverse():
4              (muA, sigmaA, omegaA) = self.getElement(e)
5              triple = [omegaA, 1 - sigmaA, muA]
6              C.setElement(e, triple)
7          return C

9      def NSdifference(self, nset):
10         if self.getUniverse() != nset.getUniverse():
11             raise ValueError("the two neutrosophic sets cannot be defined on
    different universe sets")
12         C = NSset(self.__universe)
13         for e in self.getUniverse():
14             (muA, sigmaA, omegaA) = self.getElement(e)
15             (muB, sigmaB, omegaB) = nset.getElement(e)
16             triple = [min(muA,omegaB), min(sigmaA,1-sigmaB), max(omegaA,muB)]
17             C.setElement(e, triple)
18         return C
```

The following code example executed interactively in the Python console illustrates the use of the two methods that have been just described.

```
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> U = NSuniverse('a','b','c')
```

```
>>> A = NSset(U, "(0.5,0.3,0.2), (0.6,0.2,0.3), (0.4,0.2,0.7)")
>>> print(A)
< a/(0.5,0.3,0.2), b/(0.6,0.2,0.3), c/(0.4,0.2,0.7) >
>>> B = NSset(U, "(0.2,0.2,0.2), (0.4,0.1,0.6), (0.8,0.3,0.1)")
>>> print(B)
< a/(0.2,0.2,0.2), b/(0.4,0.1,0.6), c/(0.8,0.3,0.1) >
>>> C = A.NScomplement()
>>> print(C)
< a/(0.2,0.7,0.5), b/(0.3,0.8,0.6), c/(0.7,0.8,0.4) >
>>> D = A.NSdifference(B)
>>> print(D)
< a/(0.2,0.3,0.2), b/(0.6,0.2,0.4), c/(0.1,0.2,0.8) >
```

To facilitate even more streamlined and intuitive use of neutrosophic set operations, especially in the interactive use of the framework, overloading was then used to redefine the special methods `__add__`, `__and__`, `__invert__`, `__sub__`, `__le__`, and `__ge__`, respectively referred to the operators $+$, $\&$, $\sim$, $-$, $<=$, and $>=$ for use with objects of type `NSset` by making them coincide with the methods `NSunion()`, `NSintersection()`, `NScomplement()`, `NSdifference()`, `isNSsubset()` and `isNSsuperset()`, thus obtaining the correspondence summarized in the following table.

	class method	symbol	operator
neutrosophic union	NSunion()	⋓	$+$
neutrosophic intersection	NSintersection()	⋒	$\&$
neutrosophic complement	NScomplement()	ℂ	$\sim$
neutrosophic difference	NSdifference()	⦸	$-$
neutrosophic subset	isNSsubset()	⋐	$<=$
neutrosophic superset	isNSsuperset()	⊇	$>=$

The following example illustrates how the methods and operators defined above can be easily and profitably used in the interactive mode by means of the Python console.

```
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> U = NSuniverse('a','b','c')
>>> A = NSset(U, "(0.5,0.3,0.2), (0.6,0.2,0.3), (0.4,0.2,0.7)")
>>> print(A)
< a/(0.5,0.3,0.2), b/(0.6,0.2,0.3), c/(0.4,0.2,0.7) >
>>> B = NSset(U, "(0.2,0.2,0.2), (0.4,0.1,0.6), (0.8,0.3,0.1)")
>>> print(B)
< a/(0.2,0.2,0.2), b/(0.4,0.1,0.6), c/(0.8,0.3,0.1) >
>>> print(A + B)
< a/(0.5,0.3,0.2), b/(0.6,0.2,0.3), c/(0.8,0.3,0.1) >
>>> print(A & B)
< a/(0.2,0.2,0.2), b/(0.4,0.1,0.6), c/(0.4,0.2,0.7) >
>>> print(~A)
< a/(0.2,0.7,0.5), b/(0.3,0.8,0.6), c/(0.7,0.8,0.4) >
>>> F = A - B
>>> print(F)
< a/(0.2,0.3,0.2), b/(0.6,0.2,0.4), c/(0.1,0.2,0.8) >
>>> print(F <= A)
True
>>> print(F == A & ~B)
True
```

6. The `NSmapping` class

The mappings between two universe sets and the main operations involving them are represented and handled through the NSmapping class that uses both the NSuniverse class and the NSset class.

For every mapping $f : \mathbb{U} \to \mathbb{V}$, the class stores the domain $\mathbb{U}$ and the codomain $\mathbb{V}$ as objects of type NSuniverse in the properties `__domain` and `__codomain` respectively, as well as the correspondence between each generic element $u \in \mathbb{U}$ and its value $f(u) \in \mathbb{V}$ by a dictionary corresponding to the property `__map = {u: f(u)` **`for`** `u` **`in`** $\mathbb{U}$`}`.
The class is briefly described in the following UML diagram.

NSmapping
__domain : object of the class NSuniverse __codomain : object of the class NSuniverse __map : dictionary with keys in domain and values in codomain
__init__(*args) : constructor with generic arguments getDomain() : returns the universe set corresponding to the domain getCodomain() : returns the universe set corresponding to the codomain getMap() : returns the dictionary containing the element-value pairs setValue(u,v) : assigns the value v to the element u getValue(u) : returns the value of the element u by the mapping getFibre(v) : returns the fibre of v as a list of elements of the domain NSimage(nset) : returns the neutrosophic image of a SVN-set by the mapping NScounterimage(nset) : returns the neutrosophic inverse image of a SVN-set __eq__() : checks if two mappings are equal overloading the == operator __ne__() : checks if two mappings are different overloading the ! = operator __str__() : returns the mapping in string format __repr__() : returns a detailed representation of the mapping

The constructor method accepts one or three arguments and allows us to define a mapping in several different ways:

- in the form NSmapping(*domain, codomain, values*) where *domain* and *codomain* are both universe sets expressed in any of the ways already seen above, namely as tuples, lists, strings, or instances of the class NSuniverse, while *values* is an enumeration of codomain values neatly corresponding to domain values which can be expressed indifferently as a tuple, list, string, dictionary or extended dictionary,

- in the form NSmapping(*values*) where *values* is either a regular Python dictionary or an extended dictionary; in this case the universe sets related to the domain and codomain will be created automatically by collecting respectively the keys and values of the dictionary passed as parameter, without repeating their values and checking that no error condition occurs,

- in the form NSmapping(*mapping*) by copying another object of the type NSmapping.

The basic steps of this method are described in the following algorithm.

Constructor method of the class `NSmapping`

Function `__init__`(*args*)**:**
  Create a dictionary *map*
  Get the *length* of *args*
  if *length* = 0 **then**
    │ Raise an Exception

  else if *length* = 1 **then**
      if *args* **is** *an object of type* `NSmapping` **then**
        │ Copies the properties in the current object

      else if *args* **is** *a dictionary* **then**
        │ Copy *args* to *map* and gets *domain* and *codomain* as keys and values of *args*,
        │   respectively

      else if *args* **is** *an extended dictionary* **then**
        │ Gets the dictionary from the string and passes it to the same constructor to
        │   obtain an object of type `NSmapping` from which to derive *domain*,
        │   *codomain* and the dictionary of correspondence *map*

  else if *length* = 3 **then**
      Attempts to take the first three parameters of *args* to assign them respectively
        to the universe sets *domain*, *codomain* and the object *values*
      if *values* **is** *a dictionary or an extended dictionary* **then**
        │ Passes *values* to the same constructor to obtain an object of type
        │   `NSmapping` and, if its domain and codomain are compatible with those
        │   passed as parameters, derive the dictionary of correspondences *map*

      else if *values* **is** *a list, a tuple or a string* **then**
        │ **if** *values* **is** *a list or a tuple* **then**
        │   │ Converts *values* to a list of strings;
        │ **else**
        │   │ Split *values* and turns it into a string list;
        │ **if** *the length of values* ≠ *cardinality of the domain* **then**
        │   │ Raise an Exception
        │ **if** *the set of values* **is not** *contained in the codomain* **then**
        │   │ Raise an Exception
        │ Neatly stores the elements of *values* as values of the *map* dictionary whose
        │   keys are the elements of the *domain*

      else
        │ Raise an Exception

  else
    │ Raise an Exception

  Stores *domain*, *domain* and *map* in the properties `__domain`, `__codomain` and `__map`
    respectively

Let us note how every possible error condition – such as recalling it with only one parameter that is not an object `NSmapping`, with a number of parameters other than one and three or, again, passing parameters that are not the two universe sets and a list of values however expressed – is intercepted in the code and reported to the client by raising an appropriate exception.

The Python code corresponding to the constructor method of this class is given below.

```python
from .ns_universe import NSuniverse
from .ns_set import NSset
from .ns_util import NSreplace, NSstringToDict, NSisExtDict


class NSmapping:

    def __init__(self, *args):
        map = dict()
        length = len(args)
        if length == 0:
            raise ValueError("constructor method must have at least one
    parameter")
        elif length == 1:
            if type(args[0]) == NSmapping:
                domain = args[0].getDomain()
                codomain = args[0].getDomain()
                map = args[0].getMap()
            elif type(args[0]) == dict:
                map = args[0]
                domain = NSuniverse(list(map.keys()))
                codomain = NSuniverse(list(set(map.values())))
            elif type(args[0]) == str:
                try:
                    map_dict = NSstringToDict(args[0])
                except:
                    raise ValueError("invalid parameter")
                nsmap = NSmapping(map_dict)
                domain = nsmap.getDomain()
                codomain = nsmap.getCodomain()
                map = nsmap.getMap()
            else:
                raise ValueError("the type of the parameter do not match those
     of the constructor method")
        elif length == 3:
            try:
                domain = NSuniverse(args[0])
            except:
                raise ValueError("the first parameter of the constructor
    method must be a universe set")
            try:
                codomain = NSuniverse(args[1])
            except:
                raise ValueError("the second parameter of the constructor
    method must be a universe set")
            values = args[2]
            card_domain = domain.cardinality()
            if type(values)==dict or NSisExtDict(values)==True:
                nsmap = NSmapping(values)
                if set(nsmap.getDomain()) != set(domain):
                    raise ValueError("the indicated domain is incompatible
    with the definition of the mapping")
                if nsmap.getCodomain().isSubset(codomain) == False:
                    raise ValueError("the indicated codomain is incompatible
    with the definition of the mapping")
                map = nsmap.getMap()
```

```
50              elif type(values) in [list, tuple, str]:
51                  if type(values) in [list, tuple]:
52                      values = [str(e) for e in values]
53                  else:
54                      sostituz = {"[": "", "]": "", "(": "", ")": "",
55                                  ",": " ", ";": " "}
56                      values = NSreplace(values, sostituz).split()
57                  if len(values) != card_domain:
58                      raise IndexError("the number of values passed does not
        coincide with the cardinality of the declared domain")
59                  values_set = set(values)
60                  codomain_set = set(codomain.get())
61                  if not values_set.issubset(codomain_set):
62                      raise ValueError("one or more values do not belong to the
        declared codomain")
63                  for i in range(card_domain):
64                      map[domain.get()[i]] = values[i]
65              else:
66                  raise ValueError("the third parameter of the constructor
        method must express a obj match")
67          else:
68              raise IndexError("the number of parameters do not match those of
        the constructor method")
69          self.__domain = domain
70          self.__codomain = codomain
71          self.__map = map
```

Note that, as in the case of the definition of the universe set constructor method, it is excluded that the third parameter, corresponding to the enumeration of values can be an object of type set since by its nature as an unordered collection of data would provide an ambiguous formulation of the mapping.

As in the case of the objects NSuniverse and NSset, for the mappings represented by objects of the type NSmapping much attention was paid to the usability and flexibility of the syntax which allows us to define mappings between universe sets in a variety of possible forms. For example, the mappings $f : \mathbb{U} \to \mathbb{V}$ between the universe sets $\mathbb{U} = \{a, b, c\}$ and $\mathbb{V} = \{1, 2\}$ and such that $f(a) = f(c) = 2$ and $f(b) = 1$ can be defined as an object of the class NSmapping in any of the following ways mutually equivalent:

- NSmapping(['a','b','c'], [1,2], [2,1,2]) by using lists,
- NSmapping(('a','b','c'), (1,2), (2,1,2)) by using tuples,
- NSmapping("a,b,c", "1;2", "2,1,2") by using strings,
- NSmapping("(a,b,c)", "{1,2}", "(2;1;2)") by using strings containing lists or tuples,
- NSmapping(['a','b','c'], (1,2), "2,1,2")) in a mixed form of lists, tuples and strings,
- NSmapping({'a':2, 'b':1, 'c':2}) by using dictionaries,

- `NSmapping("'a':2, 'b':1, 'c':2")` by using a string that contains a matching of values in a dictionary format,
- `NSmapping("a->2, b->1, c->2")` by using an extended dictionary with the arrow notation,
- `NSmapping("a|->2, b|->1, c|->2")` by using an extended dictionary with the the "maps to" notation,
- `NSmapping("'a'->2, b|->1; (c->2)")` by using an extended dictionary in a mixed form,
- `NSmapping("a,b,c", "1,2", "a->2, b->1, c->2")` by declaring domain and codomain and using an extended dictionary,
- `NSmapping("a,b,c", "1,2", "c->2, a|->2; b|->1")` by declaring domain and codomain and using an extended dictionary in a mixed form and without a precise order,

as well as in different combinations of them or, again, by preliminarily defining one or both of the universe sets in any of the forms already seen above, by setting, for example, `U=NSuniverse("{a,b,c}")` and `V=NSuniverse((1,2)")`, so that they can be used later in the definition of the mapping in the form like:

- `NSmapping(U, V, [2,1,2])`, or
- `NSmapping(U, V, "a->2, b->1, c->2")`.

Three basic methods called `getDomain()`, `getCodomain()` and `getMap()` respectively return us the domain and codomain of the mapping as objects of type `NSuniverse` as well as the dictionary containing all the element-value pairs that define the mapping.

```
1    def getDomain(self):
2        return self.__domain.get()

4    def getCodomain(self):
5        return self.__codomain.get()

7    def getMap(self):
8        return self.__map
```

The method `setValue(`u`,`v`)` assigns a single value v by the current mapping to a specific element u of the domain.

```
1    def setValue(self, u, v):
2        u = str(u)
3        v = str(v)
4        if u not in self.__domain.get():
5            raise IndexError('non-existent element in the domain of the
     mapping')
6        if v not in self.__codomain.get():
```

```
7                raise IndexError('non-existent element in the codomain of the
    mapping')
8            self.__map[u] = v
```

Instead, the method getValue(u) returns the value corresponding to an element u of the domain by the current mapping.

```
1      def getValue(self, u):
2          u = str(u)
3          if u not in self.__domain.get():
4              raise IndexError('non-existent element in the domain of the
    mapping')
5          return self.__map[u]
```

In order to be able to easily print on the screen objects of type NSmapping in text format and to provide a complete representation of them, the special methods __str__ and __repr__ were overloaded as follows.

```
1      def __str__(self):
2          unvwidth = 28
3          totwidth = unvwidth*2 + 8
4          s = f"\n {str(self.__domain):>{unvwidth}}   ->   {str(self.__codomain)
    :<{unvwidth}}\n"+"-"*totwidth+"\n"
5          for e in self.__domain:
6              s += f" {e:>{unvwidth}}  |->  {self.__map[e]:<{unvwidth}}\n"
7          return s

9      def __repr__(self):
10         return f"Neutrosophic mapping: {str(self)}"
```

Thanks to the redefinition by overloading of the special methods __eq__ and __ne__ we can apply the operators of equality == and diversity != directly to objects of type NSmapping.

```
1      def __eq__(self, g):
2          if self.getDomain() != g.getDomain() or self.getCodomain() != g.
    getCodomain():
3              return False
4          else:
5              equal = True
6              for e in self.getDomain():
7                  if self.getValue(e) != g.getValue(e):
8                      equal = False
9                      break
10             return equal

12     def __ne__(self, g):
13         different = not (self == g)
14         return different
```

The following code executed in interactive mode in the Python console illustrates the use of the methods just described.

```
>>> from pyns.ns_universe import NSuniverse
>>> from pyns.ns_set import NSset
>>> from pyns.ns_mapping import NSmapping
>>> U = NSuniverse("a,b,c")
>>> V = NSuniverse(1,2)
>>> f = NSmapping(U,V, (2,1,2))
>>> print(f)
                { a, b, c }   ->   { 1, 2 }
------------------------------------------------------------
                        a  |->  2
                        b  |->  1
                        c  |->  2
>>> print(f.getValue('a'))
2
>>> g = NSmapping("a->2  b->1  c->2")
>>> print( f==g )
True
>>> print(g)
                { a, b, c }   ->   { 1, 2 }
------------------------------------------------------------
                        a  |->  2
                        b  |->  1
                        c  |->  2
>>> print(h.getDomain())
{ a, b, c }
>>> print(h.getCodomain())
{ 1, 2 }
>>> print(f.getMap())
{'a': '2', 'b': '1', 'c': '2'}
>>> h = NSmapping("a,b,c", "1,2", "a->2  b->1  c->2")
>>> print( f==h )
True
```

The method `getFibre(v)` returns the fibre of an element v of the codomain by the current mapping $f : \mathbb{U} \to V$, that is, the set of all elements of the domain whose value is v, i.e. $f^{-1}(\{v\}) = \{u \in \mathbb{U} : f(u) = v\}$. The corresponding code is given below.

```python
1    def getFibre(v):
2        v = str(v)
3        if v not in self.__codomain.get():
4            raise IndexError('non-existent element in the codomain of the
     mapping')
5        fibre = list()
6        for e in self.__map:
7            if self.__map[e] == v:
8                fibre.append(e)
9        return fibre
```

Definition 6.1. [11, 22] Let $f : \mathbb{U} \to \mathbb{V}$ be a mapping between two universe sets $\mathbb{U}$ and $\mathbb{V}$, and $\widetilde{A} = \langle \mathbb{U}, \mu_A, \sigma_A, \omega_A \rangle$ be a SVN-set over $\mathbb{U}$. The *neutrosophic image* of $\widetilde{A}$ by f, denoted by

$\widetilde{f}\left(\widetilde{A}\right)$, is the SVN-set over $\mathbb{V}$ defined by:

$$\widetilde{f}\left(\widetilde{A}\right) = \left\langle \mathbb{V}, \mu_{f(A)}, \sigma_{f(A)}, \omega_{f(A)} \right\rangle$$

where the mappings $\mu_{f(A)} : \mathbb{V} \to I$, $\sigma_{f(A)} : \mathbb{V} \to I$ and $\omega_{f(A)} : \mathbb{V} \to I$ are defined respectively by:

$$\mu_{f(A)}(v) = \begin{cases} \sup\limits_{u \in f^{-1}(\{v\})} \mu_A(u) & \text{if } f^{-1}(\{v\}) \neq \emptyset \\ 1 & \text{otherwise} \end{cases},$$

$$\sigma_{f(A)}(v) = \begin{cases} \sup\limits_{u \in f^{-1}(\{v\})} \sigma_A(u) & \text{if } f^{-1}(\{v\}) \neq \emptyset \\ 1 & \text{otherwise} \end{cases},$$

$$\omega_{f(A)}(v) = \begin{cases} \inf\limits_{u \in f^{-1}(\{v\})} \omega_A(u) & \text{if } f^{-1}(\{v\}) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

for every $v \in \mathbb{V}$.

The method `NSimage(self, nset)` returns the image of a SVN-set *nset* over the domain by the current mapping. The corresponding code is given below.

```
def NSimage(self, nset):
    result = NSset(self.__codomain)
    for v in self.getCodomain():
        fibre = self.getFibre(v)
        if fibre == []:
            triple = [1,1,0]
        else:
            mu_values = list()
            sigma_values = list()
            omega_values = list()
            for u in fibre:
                mu_values.append(nset.getMembership(u))
                sigma_values.append(nset.getIndeterminacy(u))
                omega_values.append(nset.getNonMembership(u))
            triple = [max(mu_values),max(sigma_values),min(omega_values)]
        result.setElement(v, triple)
    return result
```

Definition 6.2. [11, 22] Let $f : \mathbb{U} \to \mathbb{V}$ be a mapping between two universe sets $\mathbb{U}$ and $\mathbb{V}$, and $\widetilde{B} = \langle \mathbb{V}, \mu_B, \sigma_B, \omega_B \rangle$ be a SVN-set over $\mathbb{V}$. The *neutrosophic inverse image* of $\widetilde{B}$ by f, denoted by $\widetilde{f}^{-1}\left(\widetilde{B}\right)$, is the SVN-set over $\mathbb{U}$ defined by:

$$\widetilde{f}^{-1}\left(\widetilde{B}\right) = \left\langle \mathbb{U}, \mu_{f^{-1}(B)}, \sigma_{f^{-1}(B)}, \omega_{f^{-1}(B)} \right\rangle$$

where the mappings $\mu_{f^{-1}(B)} : \mathbb{U} \to I$, $\sigma_{f^{-1}(B)} : \mathbb{U} \to I$ and $\omega_{f^{-1}(B)} : \mathbb{U} \to I$ are defined respectively by:

$$\mu_{f^{-1}(B)}) = \mu_B \circ f, \quad \sigma_{f^{-1}(B)} = \sigma_B \circ f, \quad \text{and} \quad \omega_{f^{-1}(B)} = \omega_B \circ f.$$

The method `NScounterimage(self, nset)` returns the counter image of a SVN-set *nset* over the codomain by the current mapping. The corresponding code is provided below.

```
def NScounterimage(self, nset):
    result = NSset(self.__domain)
    for u in self.getDomain():
        value = self.getValue(u)
        triple = nset.getElement(value)
        result.setElement(u, triple)
    return result
```

The following code executed in interactive mode in the Python console summarizes and explicates the use of the methods just described along with those already seen in the other two classes.

```
>>> U = NSuniverse("a,b,c,d,e")
>>> V = NSuniverse(1,2,3,4)
>>> f = NSmapping(U, V, (1,3,1,2,1))
>>> print(f)
        { a, b, c, d, e }   ->   { 1, 2, 3, 4 }
---------------------------------------------------------------
                          a  |->   1
                          b  |->   3
                          c  |->   1
                          d  |->   2
                          e  |->   1
>>> A = NSset(U, "(0.7,0.3,0.1), (0.4,0.6,0.9), (0,0,1), (0.1,0.4,0.5), (0.2,0.2
    ,0.3)")
>>> print(f"{A:t}")
        |   membership  |  indeterminacy | non-membership |
    ---------------------------------------------------------------
 a          |          0.7 |            0.3 |            0.1 |
 b          |          0.4 |            0.6 |            0.9 |
 c          |          0.0 |            0.0 |            1.0 |
 d          |          0.1 |            0.4 |            0.5 |
 e          |          0.2 |            0.2 |            0.3 |
    ---------------------------------------------------------------
>>> B = f.NSimage(A)
>>> print(B)
< 1/(0.7,0.3,0.1), 2/(0.1,0.4,0.5), 3/(0.4,0.6,0.9),
4/(1.0,1.0,0.0) >
>>> C = f.NScounterimage(B)
< a/(0.7,0.3,0.1), b/(0.4,0.6,0.9), c/(0.7,0.3,0.1),
d/(0.1,0.4,0.5), e/(0.7,0.3,0.1) >
>>> print(A.isNSsubset(C))
True
```

7. **Conclusions**

In this paper we have presented `PYNS`, an open source framework developed in Python and consisting of three distinct classes designed to manipulate in a simple and intuitive way symbolic representations of neutrosophic sets over universes of various types as well as mappings between them.

The codebase of this framework, currently comprising approximately 1200 lines of code, empowers us with the capability to seamlessly define, represent, and manipulate universe sets, neutrosophic sets, and functions operating between neutrosophic sets. This is facilitated through a comprehensive set of operations, including neutrosophic union, neutrosophic intersection, neutrosophic difference, as well as the computation of image and back-image of a neutrosophic set by means of a function, among others. These operations operate at various levels, impacting the values of the membership, indeterminacy and non-membership degree of each individual element.

The capabilities offered by this framework extend and generalize previous attempts to provide software solutions for the manipulation of neutrosophic sets already undertaken in recent years by several authors such as. Salama et al. [21], Saranya et al. [23], El-Ghareeb [7], Topal et al. [29] and Sleem [26].

Furthermore, the modular structure of `PYNS` not only facilitates interactive usage for experimentation and counterexample searches within the neutrosophic domain, making efficient use of a simple and intuitive notation, but also enables easy integration into more complex Python projects that can take advantage of robust and extensively tested methods for operations on neutrosophic sets that this framework provides.

Both the code and the underlying data structures of the three classes `NSuniverse`, `NSset` and `NSmapping` with particular regard to their properties and methods have been explained in detail in the previous sections and also concrete examples of using the introduced objects and methods have been given.

The attention given to the usability of these classes and the extensive documentation provided with a rich assortment of examples and use cases, gives us confidence that, in addition to being used for the exploration of uncertain data and practical applications, it can be the subject of further study and expansion opening up new research perspectives in various scientific and applied disciplines that use the tools of neutrosophic set theory. In particular, the authors believe that interesting developments in the medical field may come from the application and extension of this framework to neutrosophic hypersoft mappings that have proven to be useful in the diagnosis of hepatitis [17] or its eventual adaptation to fuzzy hypersoft mappings [2] which have proven to be useful in the diagnosis of HIV and tuberculosis [3, 18].

The complete Python framework `PYNS` including the source code of all the classes described in this paper as well as a selection of example programs that use them are available at the url github.com/giorgionordo/pythonNeutrosophicSets.

Acknowledgment

References

1. Abobala M., Hatip A. An Algebraic Approach to Neutrosophic Euclidean Geometry. *Neutrosophic Sets and Systems* **43** (1), pp. 114-123, 2021.

2. Ahsan M., Saeed M., Rahman A.U. A Theoretical and Analytical Approach for Fundamental Framework of Composite mappings on Fuzzy Hypersoft Classes. *Neutrosophic Sets and Systems* **45**, pp. 268-285, 2021.

3. Ahsan M., Saeed M., Mehmood A., Saeed M.H., Asad J. The Study of HIV Diagnosis Using Complex Fuzzy Hypersoft Mapping and Proposing Appropriate Treatment. *IEEE Access*, **9**, pp. 104405-104417, 2021.

4. Atanassov K.T. Intuitionistic Fuzzy Sets. *Fuzzy Sets and Systems* **20** (1), pp. 87-96, 1986.

5. Broumi S., Talea M., Bakali A., Smarandache F. Single valued neutrosophic graphs. *Journal of New theory* **10**, pp. 86-101, 2016.

6. Broumi S., Dhar M., Bakhouyi A., Bakali A., Talea M. Medical Diagnosis Problems Based on Neutrosophic Sets and Their Hybrid Structures: A Survey. *Neutrosophic Sets and Systems* **49** (1), pp. 1-18, 2022.

7. El-Ghareeb H.A. Novel Open Source Python Neutrosophic Package. *Neutrosophic Sets and Systems* **25**, pp. 136-160, 2019.

8. Elhassouny A., Idbrahim S., Smarandache F. Machine learning in Neutrosophic Environment: A Survey. *Neutrosophic Sets and Systems* **28** (1), pp. 58–68, 2019.

9. Guo Y., Cheng H.D. New neutrosophic approach to image segmentation. *Pattern Recognition* **42**, pp. 587-595, 2009.

10. Kunwar D., Singh J., Smarandache F. Neutrosophic statistical techniques to find migration pattern in Jaipur. *Octogon Mathematical Magazine* **26** (2), pp. 583-592, 2018.

11. Latreche A., Barkat O., Milles S., Ismail F. Single valued neutrosophic mappings defined by single valued neutrosophic relations with applications. *Neutrosophic Sets and Systems* **32**, pp. 203-220, 2020.

12. Mehmood A., Nadeem F., Nordo G., Zamir M., Park C.,Kalsoom H., Jabeen S., Khan M.Y. Generalized neutrosophic separation axioms in neutrosophic soft topological spaces. *Neutrosophic Sets and Systems* **32** (1), pp. 38–51, 2020.

13. Mehmood A., Nadeem F., Park C., Nordo G., Kalsoom H., Rahim Khan M., Abbas N. Neutrosophic soft alpha-open set in neutrosophic soft topological spaces. *Journal of Algorithms and Computation* **52**, pp. 37–66, 2020.

14. Mondal K, Pramanik S. Neutrosophic Decision Making Model of School Choice. *Neutrosophic Sets and Systems* **7** (1), pp. 62-68, 2015.

15. Nordo G., Mehmood A., Broumi S. Single Valued Neutrosophic Filters. *International Journal of Neutrosophic Science* **6**, pp. 8-21, 2020.

16. Saber Y., Alsharari F., Smarandache F. On Single-Valued Neutrosophic Ideals in Şostak Sense. *Symmetry* **12** (2):193, 2020.

17. Saeed M., Ahsan M., Saeed M.H., Mehmood A., Abdeljawad T. An Application of Neutrosophic Hypersoft Mapping to Diagnose Hepatitis and Propose Appropriate Treatment. *IEEE Access*, **9**, pp. 70455-70471, 2021.

18. Saeed M., Ahsan M., Saeed M.H., Rahman A.U., Mazin A.M., Nedoma J., Martinek R. An algebraic modeling for tuberculosis disease prognosis and proposed potential treatment methods using fuzzy hypersoft mappings. *Biomedical Signal Processing and Control*, **80** (1), 104267, 2023.

19. Salama A.A., Alagamy H. Neutrosophic Filters. *International Journal of Computer Science Engineering and Information Technologt Research* **3** (1), pp. 307-312, 2013.

20. Salama A., El-Ghareeb H.A.. Manie A.M., Smarandache F. Introduction to develop some software programs for dealing with neutrosophic sets. *Neutrosophic Sets and Systems* **3**, pp. 51-52, 2019.

21. Salama A., Abd el-Fattah M., El-Ghareeb H.A.. Manie A.M. Design and Implementation of Neutrosophic Data Operations Using Object Oriented Programming. *International Journal of Computer Application* **4** (5), pp. 163-175, 2014.

22. Salama A.A., Smarandache F., Kromov V. Neutrosophic Closed Set and Neutrosophic Continuous Functions. *Neutrosophic Sets and Systems* **4**, pp. 4-8, 2014.

23. Saranya S., Vigneshwaran M., Jafari S. C# Application to Deal with Neutrosophic $g\alpha$-Closed Sets In Neutrosophic Topology. *Application and Applied Mathematics* **15** (1), pp. 226-239, 2020.

24. Schweizer P. Uncertainty: two probabilities for the three states of neutrosophy. *International Journal of Neutrosophic Science* **2** (1), pp. 18-26, 2020.

25. Sharma M., Kandasamy I., Vasantha W.B. Comparison of neutrosophic approach to various deep learning models for sentiment analysis. *Knowledge-Based Systems* **223**, pp. 1-14, 2021.

26. Sleem A., Abdel-Baset M., El-henawy I. PyIVNS: A python based tool for Interval-valued neutrosophic operations and normalization. *SoftwareX* **12**, pp. 1-7, 2020.

27. Smarandache F. A Unifying Field in Logics. Neutrosophy: Neutrosophic Probability, Set and Logic. Rehoboth: American Research Press, 1999.

28. Smarandache F. Introduction to Neutrosophic Statistics. USA: Sitech & Education Publishing, 2014.

29. Topal S., Broumi S., Bakali A., Talea M., Smarandache F. A Python Tool for Implementations on Bipolar Neutrosophic Matrices. *Neutrosophic Sets and Systems* **28**, pp. 138-161, 2019.

30. Wang H., Smarandache F., Zhang Y.Q., Sunderraman R. Single Valued Neutrosophic Sets. *Technical Sciences and Applied Mathematics*, pp. 10-14, 2012.

31. Zadeh L.A. Fuzzy Sets, *Information and Control* **8** (3), pp. 338-353, 1965.

32. Zhang M., Zhang L. Cheng H.D. A neutrosophic approach to image segmentation based on watershed method. *Signal Processing* **90**, pp. 1510-1517, 2010.