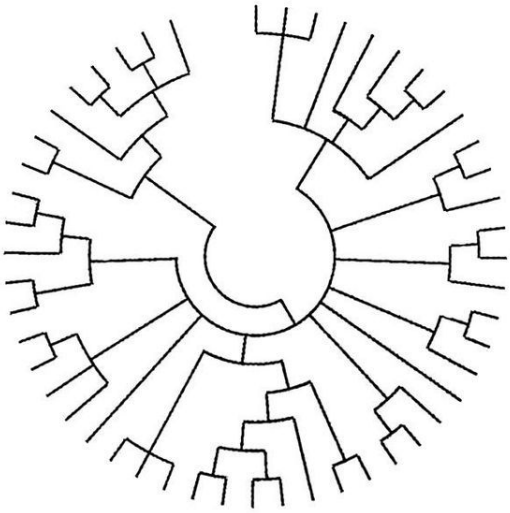
DOCUMENTATION FOR THE PYTHON LIBRARY PEDIGRAD.PY

Rémy Tuyéras
rtuyeras@mit.edu

Version 2019.09.24



Typeset in LaTeX 2$_\varepsilon$.

Contents

Chapter 1

Introduction

1.1. About pedigrads and `Pedigrad.py`

Pedigrads are mathematical tools that were initially created to model genetic mechanisms (see [1, 2, 3]). Mathematically, pedigrads are cone-preserving functors going from a certain class of limit sketches to a given category of values. Different aspects of biology can be encoded depending on the category of values considered. The present python library – `Pedigrad.py` – provides tools modelling the content described in paper [2].

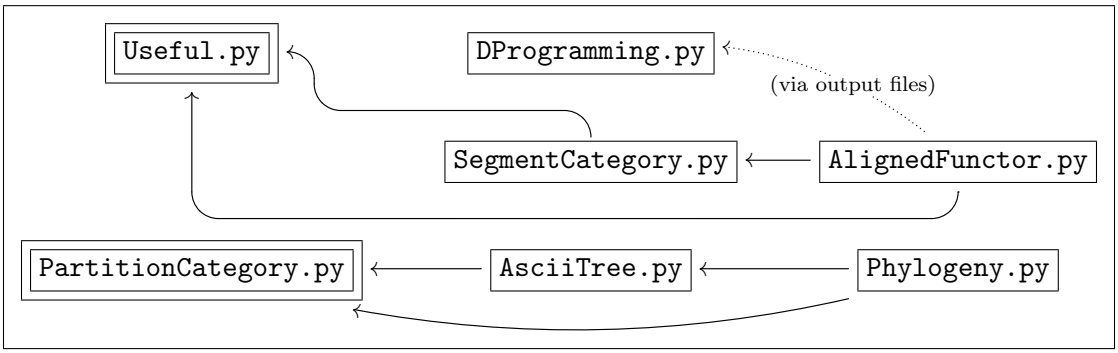
Earlier versions of this documentation focused on pedigrads in the category of partitions to model mechanistic aspects of phylogenetics. Hence, parts of this documenation comes with modules on partitions and evolutionary trees.

1.2. About this documentation

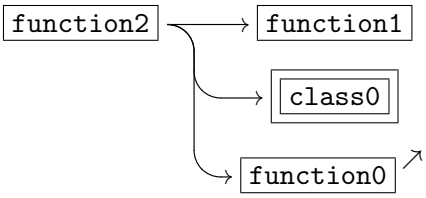
The present book contains a tutorial (see Chapter 2) explaining how to use the various methods and classes contained in `Pedigrad.py` as well as a description of the (importable and non-importable) functions and classes of its sub-modules

- `Useful.py` (see Chapter 3)
- `DProgramming.py` (see Chapter 4)
- `SegmentCategory.py` (see Chapter 5)
- `AlignedFunctor.py` (see Chapter 6)
- `PartitionCategory.py` (see Chapter 7)
- `AsciiTree.py` (see Chapter 8)
- `Phylogeny.py` (see Chapter 9)

The dependencies between the different chapters is displayed below.



Simiarly, descriptions of functions given in this documentation will usually start with dependency flow charts showing the modules that they use. Below, we give the example of such a flow chart.



As can be seen above, these flow charts will use three different types of boxes:

1) boxes with no specific decoration will usually frame items (such as classes and functions) that belong to the module of the described function;
2) double boxes will frame the name of intermediate items that do not have dependencies with other items (final item);
3) boxes with arrows in the top-right corner will frame items that are defined in modules external to that of the described function;

We will also give examples and demonstrations within editor mode windows or console mode windows. The editor mode will be used to describe the code of functions and will look as follows.

```
1  class MyClass:
2    """
3    This is a comment
4    """
5    def __init__(self,arg0,arg1):  #This is the constructor of the class
6      """
7      Another comment about the code
8      """
```

The console mode will be used for examples and will look as follows.

```
>>> P.obj
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13]
```

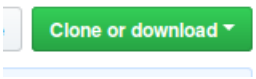
1.3. Acknowledgments

I would like to thank Maxim Wolf and Carles Boix for interesting discussions about DNA and genetics. I would also like to thank Maxim Wolf for answering many of my questions and giving me some of his time.

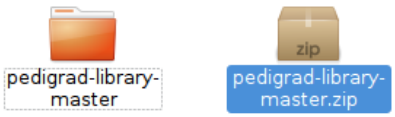
Tutorial

2.1. Installation and preparation

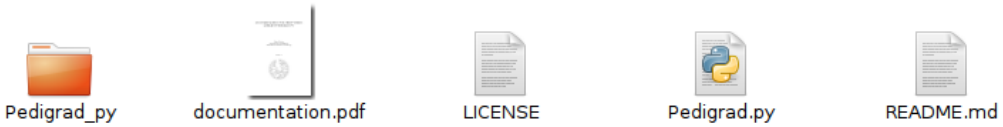
To install the library, first download the package by clicking on the green button on the right of the screen at https://github.com/remytuyeras/pedigrad-library .



The downloaded package should be a compressed file named `pedigrad-library-master.zip`. Create a new directory in which you can copy and extract the compressed file using your favorite extraction application.



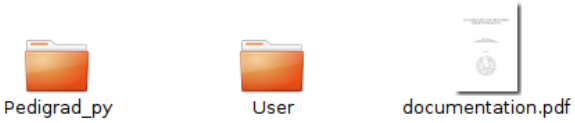
Enter the extracted directory `pedigrad-library-master`. Its inside should contain the following files.



In this tutorial, we will create four files:

- ▷ two multiple sequence alignment files (usually with an extension `.fa` or `.fasta`);
- ▷ a file in which a pre-ordered set will be specified (preferably with an extension `.yml`);
- ▷ a python file `main.py` in which we will write python functions and use the library.

To do this properly, create a new directory in `pedigrad-library-master`, call it `User`, and copy the file `Pedigrad.py` in `User`.



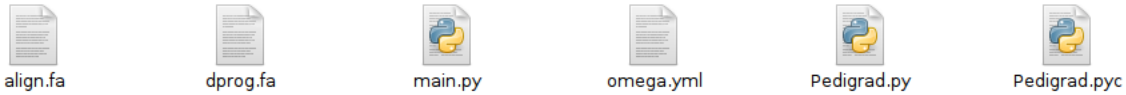
Now, open the file `Pedigrad.py` that you pasted in the folder `User`: you will be able to see several instances of the function `sys.path.insert` in which paths appear in the second argument.

```
1  import sys
2  sys.path.insert(0, "Pedigrad_py/Useful/")
3  from Useful import *
```

Add the text `../` at the beginning of every path passed to the function `sys.path.insert`, as shown below.

```
1  import sys
2  sys.path.insert(0, "../Pedigrad_py/Useful/")
3  from Useful import *
```

Once the paths are all updated, create four new files with the names `align.fa`; `dprog.fa`; `main.py`; and `omega.yml` in the directory `User`. The inside of the directory `User` should look as follows, where the file `Pedigrad.pyc` will appear later, after compilation.



Open the file `main.py` and insert the following piece of code.

main.py

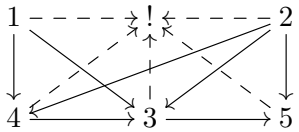
```
1  from Pedigrad import *
```

We are now ready to use the library – proceed to section 2.2.

2.2. Pre-ordered sets

This section demonstrates the use of the class `PreOrder` (section 5.1), which belongs to the module `SegmentCategory.py` (section 5). The class `PreOrder` allows the user to load a pre-ordered structure specified in a text file to the Random-Access Memory (RAM). For the sake of making this tutorial as exhaustive as possible, we will consider the pre-ordered set pictured in (2.1), which will allow us to illustrate various cases. In diagram (2.1), every (dashed or non-dashed) arrow $x \to y$ represents a relation of the form $x > y$ in the pre-order structure.

(2.1)



For this tutorial, we specify such a pre-ordered structure by inserting the following piece of code in the `omega.yml` (for the syntax, see the grammar rules shown in section 5.1.3.1).

omega.yml

```
1  !obj:
2    - 1;
3    - 2;
4    - 3;
5    - 4;
6    - 5;
7  rel:
8    - 2, 3 > 5;
9    - 1, 2 > 3, 4;
10   - 4 > 3;
```

Note that the minimal object ! appearing in diagram (2.1) is specified through the symbol `!` preceding the key word `obj:`.

We can load the pre-order structure contained in `omega.yml` by creating an item of the class `PreOrder` relative to the file `omega.yml` (see the example given below). The class `PreOrder` is equipped with three objects: `.relations`, `.mask` and `.cartesian`. Let us what these do by calling them in the file `main.py` as follows.

main.py

```
3 Omega = PreOrder("omega.yml")
4 print Omega.relations
5 print Omega.mask
6 print Omega.cartesian
```

The object `Omega.relations` is a list of lists in which the relations specified in `omega.yml` are organized in the form of (non-transitive) down-closures. More specifically, every list in `Omega.relations` is associated with a particular object of the pre-ordered set, located at the beginning of the list, and contains all those objects that are less than or equal to this particular object based on the relation specified in `omega.yml`.

The object `Omega.mask` contains a Boolean value specifying whether the file `omega.yml` creates a formal minimal object through the use of the key word `!obj:`.

Finally, the object `Omega.cartesian` is an integer specifying whether the pre-ordered set is the Cartesian product of another pre-ordered set. If this integer is 0, then the Cartesian structure is not relevant to the `PeOrder` item. Compiling the file `main.py` should give the output.

```
[['1', '4', '3'], ['2', '5', '3', '4'], ['3', '5'], ['4', '3'], ['5']]
True
0
```

As mentioned earlier, the down-closures contained in `Omega.relations` are only based on the relations specified in `omega.yml` and may hence lack certain relations that can only be deduced from transitivity. One can add these relations to the lists of `Omega.relations` by calling the method `.closure`, as shown below.

main.py

```
7 Omega.closure()
8 print "--after:"
9 print Omega.relations
```

As can now be seen on the standard output, after compilation, the list `Omega.relations` has been completed with the relations `1>5` and `3>5`, which can only be deduced by transitivity from the relations given in `omega.yml`.

```
--after:
[['1', '4', '3', '5'], ['2', '5', '3', '4'], ['3', '5'], ['4', '3', '5'],
['5']]
```

The class `PreOrder` is also equipped with a method `.geq` (short for *greater than or equal to* – see section 5.1.5). Let us have a look at the behavior of this method on objects that either belong to, or do not belong to the pre-order structure of `omega.yml`.

main.py

```
10 print "--relation:"
11 print Omega.geq("1","2")
12 print Omega.geq("2","1")
13 print Omega.geq("4","3")
14 print Omega.geq("?","3")
15 print Omega.geq("?","?")
```

As can be seen, the function `Omega.geq` can only return `True`when its inputs are elements of the pre-ordered set and are comparable. In any other case, the output is `False`.

```
--relation:
False
False
True
False
False
```

In addition to the method `.geq`, the class `PreOrder` possesses an infimum function `.inf` (section 5.1.6). By definition, for every pair of elements in the pre-ordered set, the method `.inf` returns the largest element of the pre-ordered set that is less than or equal to the two input elements. Let us see what the following lines of code give us.

main.py

```
16 print "--infimum:"
17 print Omega.inf("2","3")
18 print Omega.inf("1","2")
19 print Omega.inf("?","3")
```

After compilation, we obtain the following text of the standard output. As expected, the infimum of `2` and `3` is `3` because the relation `2 > 3` holds in `omega.yml`. Similarly, the infimum of `1` and `2` is `4` because `4` is the maximum of the values `4`, `3`, and `5`, which constitute the set of lower bounds of `1` and `2` (see diagram (2.1)). Finally, if the infimum of two values is not well-defined, then the method returns the value stored in `Omega.mask`, as shown below for the last call of the method `.inf`.

```
--infimum:
3
4
True
```

Finally, as a set, a pre-ordered set should also equipped with a relation *belong to* (*i.e.* $\in$) to check whether a given element belongs to the pre-ordered set. This relation is available for `PreOrder` item through the method `.presence` (section 5.1.7). We can, for instance, try the following presence tests.

main.py

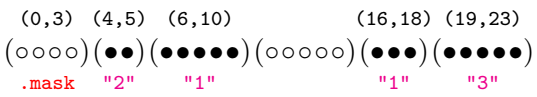
```
20 print "--presence test:"
21 print Omega.presence("3")
22 print Omega.presence("1")
23 print Omega.presence("?")
```

As expected (see output below), compiling `main.py` shows that `3` and `1` are elements of `omega.yml` and `?` is not.

```
--presence test:
True
True
False
```

2.3. Segments and morphisms of segments

The present section demonstrates the use of the classes `SegmentObject` (section 5.2) and `MorphismOfSegments` (section 5.3), which are both subclasses of the class `CategoryItem` (section 3.1). The class `SegmentObject` allows the user to generate segments as defined in [2] and the class `MorphismOfSegments` allows the user to construct morphisms between these segments (see [2, section 2]). For this library, the specification of a segment requires a topology and a color map. The former is given by a list of pairs of integers specifying successive intervals in the segments (called *patches* in [2]) and the latter is specified through a list of objects that should (but may not necessarily) belong to a `PreOrder` item (see section 2.2). In the following picture, the topology is specified above the segment while the color map is shown below the patches.



Note that the patch `(11,15)` is missing in the topology. This will often happen for 'masked' patches, that is to say patches that we want to associate with the minimal object of the `PreOrder` item, in order to save space in the RAM.

Let us now show how one can construct a segment with the class `SegmentObject`. First, insert the following piece of code in the file `main.py`

main.py

```
24
25  print "\n------------------------\n"
26
27  t = list()
28  c = list()
29  for i in range(20):
30    t = t + [(i,i)]
31    if i%15 in [11]:
32      c = c + ["?"]
33    elif i%2 in [0]:
34      c = c + ["1"]
35    elif i%2 in [1]:
36      c = c + ["2"]
```

The list `t` defines a topology while the list `c` defines a color map. We can then use these two lists two create a segment as follows.

main.py

```
37  s = SegmentObject(20,t,c)
38  sys.stdout.write("s = ")
39  s.display()
40  print "colors = "+str(s.colors)
```

The integer 20 given in the first argument of the constructor of `SegmentObject` sets the length of the segment. This length conveniently fits the coverage of the topology `t` that we defined earlier. Compiling the previous code will display the following text on the standard output.

```
------------------------

s = (o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
colors = [‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘?’, ‘1’,
‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’]
```

If we changed the length of the segment to 25, then we would obtain the following alternative output, where the last five nodes are not colored (or rather masked).

```
s = (o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|ooooo)
colors = [‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘?’, ‘1’,
‘2’, ‘1’, ‘2’, ‘1’, ‘2’, ‘1’, ‘2’]
```

However, changing the length to a high number, say 250, will not display the whole sequence of nodes on the standard output, mostly if these nodes are masked. Instead, the method `.display` will turn most of the masked nodes into a count, as shown below (the same would happen at the beginning of the segment if the index of the first node was greater than 12) .

```
s = (o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o-228-o)
```

Now, the idea is to use the segment `s` to construct more exotic segments. For instance, we can use the method `.merge` (section 5.2.6) to merge patches within the topology of `s`. The color of every new patch is defined as the infimum value of the colors of the merged patches.

In the following piece of code, we merge the patches of `s` according to 3 different tiling patterns. The triple `(0,3,9)` indicates that we will merge every set of successive triple of (non-masked) patches from the (non-masked) patch of index 0 to the (non-masked) patch of index 9, the triple `(10,2,14)` indicates that we will merge every set of successive pairs of (non-masked) patches from the (non-masked) patch of index 10 to the (non-masked) patch of index 14, and the triple `(15,3,19)` indicates that we will merge every set of successive triples of (non-masked) patches from the (non-masked) patch of index 15 to the (non-masked) patch of index 19.

main.py

```
42 s1 = s.merge([(0,3,9),(10,2,14),(15,3,19)],Omega.inf)
43 sys.stdout.write("s1 = ")
44 s1.display()
45 print "topology = "+str(s1.topology)
46 print "colors = "+str(s1.colors)
```

Compiling outputs the following text, where we can see the newly merged patches. The new colors generated through this process are either `4 = Omega.inf(1,2)` or the Boolean value `Omega.mask = Omega.inf(1,?)`, which leads to masking the associated patch.

```
s1 = (ooo|ooo|ooo|o|oo|oo|o|ooo|oo)
topology = [(0, 2), (3, 5), (6, 8), (9, 9), (12, 13), (14, 14), (15, 17),
(18, 19)]
colors = [‘4’, ‘4’, ‘4’, ‘2’, ‘4’, ‘1’, ‘4’, ‘4’]
```

In addition of merging patches, we can also ‘remove’ patches from the topology of a segment, which will lead to consider the patch as masked. We do so by using the method `.remove` and giving it the indices of the patches to be removed. Note that the method `.remove` can be quite handy when one needs to display various parts of a very long segments in which the information is distributed sparsely.

To illustrate the use of the method `.remove` , let us insert the following piece of code in the file `main.py`.

main.py

```
48 s2 = s.merge([(0,3,9),(10,2,14)],Omega.inf)
49 sys.stdout.write("s2 = ")
50 s2.display()
51
52 s2 = s2.remove([5,23])
53 sys.stdout.write("s2 = ")
54 s2.display()
```

To make our example slightly more interesting, let us also augment the length of the segment `s2` by 4 nodes, as shown below.

```
56 s2.domain = s2.domain+4
57 sys.stdout.write("s2 = ")
58 s2.display()
59 print "colors = "+str(s2.colors)
```

Compiling the file `main.py` gives the following additional text on the standard output.

```
s2 = (ooo|ooo|ooo|o|oo|oo|o|o|o|o|o|o)
s2 = (ooo|ooo|ooo|o|oo|oo|o|o|o|o|o|o)
s2 = (ooo|ooo|ooo|o|oo|oo|o|o|o|o|o|oooo)
colors = ['4', '4', '4', '2', '1', '1', '2', '1', '2']
```

We can create another segment by appending the following similar piece of code to the file `main.py`.

main.py

```
61 s3 = s.merge([(0,3,9),(10,2,14)],Omega.inf)
62 sys.stdout.write("s3 = ")
63 s3.display()
64 s3 = s3.remove([1,5,6,8,10])
65 sys.stdout.write("s3 = ")
66 s3.display()
67 s3.topology = s3.topology+[(24,24)]
68 s3.colors = s3.colors+["5"]
69
70 s3.domain = s3.domain+5
71 sys.stdout.write("s3 = ")
72 s3.display()
73 print "colors = "+str(s3.colors)
```

Compiling `main.py` adds the text to the standard output displayed below; note how the value in `s3.domain` restricts the display of the patch `(24,24)` in the second version of `s3`, but not in its last version.

```
s3 = (ooo|ooo|ooo|o|oo|oo|o|o|o|o|o|o)
s3 = (ooo|ooo|ooo|o|oo|oo|oo|o|o|o|o)
s3 = (ooo|ooo|ooo|o|oo|oo|oo|o|o|o|ooooo|o)
colors = ['4', '4', '2', '1', '2', '2', '5']
```

Let us now move on constructing morphisms between segments (see [2, section 2.5]). In particular, we want to construct a morphism between our previous two segments `s2` and `s3`. To do so, let us add the following piece of code to `main.py`, where the parameter `"id"` is treated as the list `range(s2.domain)`, which can be seen as a trivial mapping `i` $\mapsto$ `i`.

main.py

```
75 m = MorphismOfSegments(s2,s3,"id",Omega.geq)
76 print m.defined
77 sys.stdout.write("s = ")
78 m.source.display()
79 sys.stdout.write("t = ")
80 m.target.display()
81 print "f0 = "+str(m.f0)
```

After compiling, we obtain the following additional text on the standard output. The value of `m.defined` tells us whether the construction of the morphism succeeded or not. The value of `m.defined` may be `False` if the morphism does not satisfy the axioms specified in [2, section 2.5]. In the case of an ill-defined morphism, the list `m.f0`, which represents the order-preserving function $f_0 : [n_0] \to [n'_0]$ used in [2, section 2.5], should be empty (see section 5.3). Note that failure to construct a morphism of segments is likely to be due to the impossibility of constructing f_0.

```
True
s = (ooo|ooo|ooo|o|oo|oo|o|o|o|o|o|o|oooo)
t = (ooo|ooo|ooo|o|oo|oo|oo|o|o|o|ooooo|o)
f0 = [0, -1, 1, 2, 3, -1, 4, -1, 5]
```

The previous example rather simple, mostly because we used the parameter `"id"` instead of specifying list `f1` that would model an interesting mapping `i` $\mapsto$ `f1[i]`. The reason is that it can sometimes be quite time consuming to specify these types of mappings by hand. Instead, we want to use more powerful tools, such the class `CategoryOfSegments`, which presented in section 2.4.

2.4. Categories of segments

The present section demonstrates the use of the class `SegmentCategory`, which models the features of a category of segments [2]. In particular, this class will allow us to be more efficient in the specification of segments (through the method `.initial`) and morphisms of segments (through the generator `.homset`). As with category of segments, an item of the class `SegmentCategory` needs to be initialized with a `PreOrder` item.

In our case, we want to use the `PreOrder` item `Omega` defined in section 2.2. We call our category of segments as follows.

main.py

```
82
83 print "\n-----------------------\n"
81
85 Seg = CategoryOfSegments(Omega)
86
```

As with categories, we can know whether there is an identity morphism between two segments. For instance, the following lines of code test whether a segment can be equal to itself. We also test an equality between the two different segments.

main.py

```
87  s = Seg.initial(18,"1")
88  s = s.merge([(2,2,8)],Omega.inf)
89
90  print Seg.identity(s,s)
91
92  t = Seg.initial(20,"1")
93  t = t.merge([(2,3,10),(15,2,18)],Omega.inf)
94
95  print Seg.identity(s,t)
```

As expected, compiling the previous lines of code displays the following output, which tells us that there is an identity between `s` and itself, but not between `s` and `t`.

```
------------------------

True
False
```

We can check that `s` and `t` are indeed different segments by adding the following piece of code to `main.py`

main.py

```
97  sys.stdout.write("s= ")
98  s.display()
99  sys.stdout.write("t= ")
100 t.display()
```

Compiling gives us the following output.

```
s= (o|o|oo|oo|oo|o|o|o|o|o|o|o|o|o)
t= (o|o|ooo|ooo|ooo|o|o|o|o|oo|oo|o)
```

If we cannot find an obvious morphism between `s` and `t`, we can always use the method `.homset` of the class `PreOrder` to generate all the existing morphisms going from the segment `s` to the segment `t` – this done in the following lines of code.

main.py

```
102 #all morphisms in homset are well-defined
103 for i,m in enumerate(Seg.homset(s,t)):
104   print str(i)+") well-defined = "+str(m.defined)
105   print "f1 = "+str(m.f1)
106   print "f0 = "+str(m.f0)
```

Compiling the file `main.py` now gives the list of the 14 morphisms of segments `s` $\to$ `t`.

```
0) well-defined = True
f1 = [0, 1, 2, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
1) well-defined = True
f1 = [0, 1, 2, 3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
2) well-defined = True
f1 = [0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
3) well-defined = True
f1 = [0, 1, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
4) well-defined = True
f1 = [0, 1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
5) well-defined = True
f1 = [0, 1, 2, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
6) well-defined = True
f1 = [0, 1, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
7) well-defined = True
f1 = [0, 1, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
8) well-defined = True
f1 = [0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
9) well-defined = True
f1 = [0, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
10) well-defined = True
f1 = [0, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
11) well-defined = True
f1 = [0, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [0, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
12) well-defined = True
f1 = [1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [1, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
13) well-defined = True
f1 = [1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [1, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
14) well-defined = True
f1 = [1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
f0 = [1, 2, 2, 3, 4, 4, 5, 6, 7, 8, 9, 9, 10, 10, 11]
```

2.5. Environment functors

The present section demonstrates the use of the class `Environment` (section 6.3), which models the features of environment functors E_b^ε, as defined in [2, Definition 3.11]. As with environment functors, an `Environment` item is equipped with a pointed set (E, ε) (*i.e.* a `PointedSet`

item – see section 6.1), a category of segments $\mathbf{Seg}(\Omega)$ (*i.e.* a `CategoryOfSegments` item – see section 5.4) and a truncation element b taken from the pre-ordered set Ω. More specifically, initializing an `Environment` item usually takes the following lines of code.

main.py

```
107
108 print "\n------------------------\n"
109
110 E = PointedSet(["-","A","C","G","T"],0)
111
112 Env = Environment(Seg,E,5,["4"]*5) #[] = white nodes
113 print Env.Seg.preorder.relations
114 print Env.pset.symbols
115 print Env.pset.point()
116 print Env.spec
117 print Env.b
```

Compiling the previous code adds the following text to the standard output.

```
------------------------

[['1', '4', '3', '5'], ['2', '5', '3', '4'], ['3', '5'], ['4', '3', '5'],
['5']]
['-', 'A', 'C', 'G', 'T']
-
5
['4', '4', '4', '4', '4']
```

As a model for a particular type of functors, an `Environment` item is equipped with a pullback operation, which we briefly illustrate before ending this section (see section 6.3 for more details).

$$\begin{array}{ccc} \{\tau\} & \overset{\subseteq}{\hookrightarrow} & \mathbf{Seg}(\Omega) \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle \mathbf{A}E_b^{\varepsilon}} \\ \mathbf{1} & \xrightarrow[\text{input}]{} & \mathbf{Set} \end{array}$$

More specifically, we can pullback any sequence of characters, taking its values in the pointed set (E, ε), through the method `.segment`. Let us consider the following example:

main.py

```
s4 = Env.segment(["A","C","G","T","T","N","C","A","-","C","T"],"1")
s4.display()
```

As can be seen, after compiling, the characters belonging to the pointed sets are associated with non-masked nodes, while characters that do not belong to the pointed set are associated with masked nodes.

(o|o|o|o|o|o|o|o|o|o|o)

2.6. Aligned functors

The present section demonstrate the use of the class `SequenceAlignment` (section 6.2), which models the features of a sequence alignment functor, as defined in [2, Definition 3.20]. In particular, we show how the type of construction illustrated in [2, Example 3.22] can be with the functions of this library.

To start with, let us consider the following sequence alignment file.

align.fa

```
>1:A:1
ACTCGATCTCTG?TCGATCGATCG
CCTATCGGATCGATC
>1:B:2
ACTCTAT?TCTATCC-ACTCATCA
CCTACTATCTCGAAA
>1:C:3
ACTCTATC----TCCGACT?ATCA
CCTACTATCTCGAAA
>1:D:1
ACTATTTA?TTTC----TTTTCTA
CCGGGCTGGGGGGGG
```

align.fa

```
>2:A:1
ACTCGATCGGATC-AT??CGATCG
CCTATCGGATCGATCG
>2:B:2
ACTCC?CCTA--CTGGGCTGCTCA
CCTACTATCTCGAAAG
>2:C:3
ACTCTCTATC?CTCTATC---TCA
CCTACTATC?CGAAAG
>2:D:1
ACTATT-??T--AC--CTTTTCTA
CCGGGCTTTTCGAAAA
>1:K:1
ACTATCGATCTTTAGCTAGTTCTA
CC-??????----GA
```

We can create a `SequenceAlignment` item from the pairwise sequence alignments stored in the file `align.fa` by using the method `.seqali`, as shown below.

main.py

```

print "\n------------------------\n"

Seqali = Env.seqali("align.fa")
```

The `SequenceAlignment` item `Seqali` is equipped with two objects `.base` and `.database` (see section 6.2). The object `.base` contains the `SegmentObject` items at which the sequences given in `align.fa` can be found in the aligned environment functor. In other words, the object `.base` contains the objects of the category B designed in [2, Example 3.22]. The object `.database` contains the lists of sequence alignments associated with each `SegmentObject` item in `.base`. In other words, it encodes a preliminary version of the mapping $T : B \to \mathbf{Set}$ constructed in [2, Example 3.22].

The following code shows how to use the objects `.base` and `.database` in order to retrieve the information contained in `align.fa`.

main.py

```
126 print "\nDatabase\n"
127
128 print Seqali.indiv
129 for i in range(len(Seqali.base)):
130   print str(i)+") color:  "+str(Seqali.base[i].colors[
    Seqali.base[i].parse ])
131   Seqali.base[i].display()
132   for j in range(len(Seqali.database[i])):
133     for k in range(len(Seqali.database[i][j])):
134       print Seqali.database[i][j][k]
135     print ""
```

Compiling gives the following output:

```
Database

['A', 'B', 'C', 'D', 'K']
0) color:  ['1', '2', True, '1', '1']
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
ACTCGATCTCTG?TCGATCGATCGCCTATCGGATCGATC
ACTCTAT?TCTATCC-ACTCATCACCTACTATCTCGAAA
masked
ACTATTTA?TTTC----TTTTCTACCGGGCTGGGGGGGG
ACTATCGATCTTTAGCTAGTTCTACC-??????----GA

1) color:  ['1', '2', True, '1', True]
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
ACTCGATCGGATC-AT??CGATCGCCTATCGGATCGATCG
ACTCC?CCTA--CTGGGCTGCTCACCTACTATCTCGAAAG
masked
ACTATT-??T--AC--CTTTTCTACCGGGCTTTTCGAAAA
masked
```

Alternatively, we can call the images of the underlying aligned environment functor by using the method `.image`. Below, we use the method `.image` to retrieve the image of the functor at the `SegmentObject` item stored in `Seqali.base[0]` and `Seqali.base[1]`.

main.py

```
137 print "\nImage\n"
138 for i in range(len(Seqali.base)):
139   print "base["+str(i)+"]"
140   Seqali.base[0].display()
141   sal = Seqali.eval(Seqali.base[0])
142   for j in range(len(sal)):
143     for k in range(len(sal[j])):
144       print sal[j][k]
145     print ""
```

Compiling the previous code gives the following text on the standard output:

```
Image

base[0]
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o
|o|o|o)
ACTCGATCTCTG?TCGATCGATCGCCTATCGGATCGATC
ACTCTAT?TCTATCC-ACTCATCACCTACTATCTCGAAA
masked
ACTATTTA?TTTC----TTTTCTACCGGGCTGGGGGGGG
ACTATCGATCTTTAGCTAGTTCTACC-??????----GA

base[1]
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o
|o|o|o|o)
```

```
ACTCGATCGGATC-AT??CGATCGCCTATCGGATCGATCG
ACTCC?CCTA--CTGGGCTGCTCACCTACTATCTCGAAAG
masked
ACTATT-??T--AC--CTTTTCTACCGGGCTTTTCGAAAA
masked
```

In addition to being able to display the images of the sequence alignment functor, we can also generate the sequence of objects of its underlying *extending category*, which is defined in [2, Definition 3.25].

main.py

```
print "\nExtending_category\n"

for i,m in Seqali.extending_category(Seqali.base[0]):
  print i
  print m.f1
  print m.f0
```

Compiling the previous lies of codes gives the sequence of all the objects of the extending category associated with the `SegmentObject` item stored in `Seqali.base[0]` (for conciseness, only the beginning and the end of the output is given)

```
Extending category

0
id
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]
1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]
...
1
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

Of course, we can do the same for the `SegmentObject` item stored in `Seqali.base[1]`.

main.py

```
154 print "\nExtending diagram\n"
155
156 for i,m in Seqali.extending_category(Seqali.base[1]):
157   print i
158   print m.f1
159   print m.f0
```

However, this time, the extending category possesses only one object.

```
Extending diagram

1
id
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]
```

2.7. Sequence alignments and the dynamic programming

The present section demonstrate the use of the class `Table` (section 4.3), models the features of a dynamic programming score table for the purpose of aligning two pair of sequences of characters. In particular, it encodes the dynamic programming algorithm that allows the type of analysis discussed in [2, Section 4] for the recognition of biological mechanisms.

The class `Table` should be used as shown in the following example. First, use the class `Sequence` (section 4.2) to initialize the table and then use the method `.incidence` to compute all the possible matches between the two sequences given to the `Table` item.

main.py

```
160
161 print "\n------------------------\n"
162
163 a = list("AGCTAGCTGA")
164 b = list("GTGGATCGATGA")
165
166 A = Sequence("a",a,"1")
167 B = Sequence("b",b,"1")
168
169 table = Table(A,B)
170 print "\nincidence"
171 table.incidence()
172 table.stdout()
```

Compiling the file `main.py` displays the following text on the standard output.

```
------------------------


incidence
.  |  .  |  G  |  T  |  G  |  G  |  A  |  T  |  C  |  G  |  A  |  T  |  G  |  A  |
.  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |
A  |  0  |  0  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |
G  |  0  |  1  |  0  |  1  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |  0  |
C  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  0  |  0  |
T  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |
A  |  0  |  0  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |
G  |  0  |  1  |  0  |  1  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |  0  |
C  |  0  |  0  |  0  |  0  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  0  |  0  |
T  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |
G  |  0  |  1  |  0  |  1  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |  0  |
A  |  0  |  0  |  0  |  0  |  0  |  1  |  0  |  0  |  0  |  1  |  0  |  0  |  1  |
```

Then, the method `.fillout` should be called to fill out the table with the scores that will be used by the dynamic programming algorithm.

main.py

```
173 print "\nfillout"
174 table.fillout()
175 table.stdout()
```

Compiling the file `main.py` now adds the following text on the standard output.

```
fillout
. | . | G | T | G | G | A | T | C | G | A | T | G | A |
. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
A | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
C | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
T | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
A | 0 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 |
G | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
C | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 | 4 |
T | 0 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 |
G | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 |
A | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 7 |
```

The method `.dynamic_programming` can now be used to store all the valid sequence alignments resulting from the previous scores in the file `"dprog.fa"`. As with the function `open`, the second argument of `.dynamic_programming` takes either `"w"`, to rewrite on the file, or `"a"`, to append the outputs to an existing file. To do so, add the following line of code to `main.py`.

main.py

```
176 table.dynamic_programming("dprog.fa",option = "w",debug = False,display
    = True)
```

Compiling gives the following data on the standard output and inside the file `prog.fa`.

```
>0:a:1
AGCTA-G--C--TGA
>0:b:1
-G-T-GGATCGATGA
>1:a:1
AGCT-AG--C--TGA
>1:b:1
-G-TG-GATCGATGA
>2:a:1
AGCTAG---C--TGA
>2:b:1
-G-T-GGATCGATGA
>3:a:1
AGCT--AG-C--TGA
>3:b:1
-G-TGGA-TCGATGA
>4:a:1
AGCT--A-GC--TGA
>4:b:1
-G-TGGAT-CGATGA
```

```
>5:a:1
AGCT--A--GC-TGA
>5:b:1
-G-TGGATCG-ATGA
>6:a:1
AGCT--A--G-CTGA
>6:b:1
-G-TGGATCGA-TGA
```

Finally, it is possible to use the file `dprog.fa` in the same fashion as we used the file `align.fa`, in section 2.6. For instance, add the following line of codes and compile.

main.py

```
177
178 print "\n------------------------\n"
179
180 E = PointedSet(["-","A","C","G","T"],0)
181
182 Env = Environment(Seg,E,2,["1"]*2) #[] = white nodes
183 Seqali = Env.seqali("dprog.fa")
184
185 print "\nImage\n"
186
187 print Seqali.indiv
188 Seqali.base[0].display()
189 sal= Seqali.eval(Seqali.base[0])
190 for j in range(len(sal)):
191   for k in range(len(sal[j])):
192     print sal[j][k]
193   print ""
```

The following result should be obtained:

```
Image

['a', 'b']
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
AGCTA-G--C--TGA
-G-T-GGATCGATGA

AGCT-AG--C--TGA
-G-TG-GATCGATGA

AGCTAG---C--TGA
-G-T-GGATCGATGA

AGCT--AG-C--TGA
-G-TGGA-TCGATGA

AGCT--A-GC--TGA
-G-TGGAT-CGATGA
```

```
AGCT--A--GC-TGA
-G-TGGATCG-ATGA

AGCT--A--G-CTGA
-G-TGGATCGA-TGA
```

Presentation of the module `Useful.py`

3.1. Description of `CategoryItem` (class)

3.1.1. Introduction. The class `CategoryItem` models the diagrammatic features of the elements of a category. This class is meant to be used as a super class. In particular, the class `CategoryItem` is equipped with an object `.level` that can be used to model the polymorphic nature functors, namely both arrows and objects (of the considered category) can be given to a function as `CategoryItem` items and can be handled differently depending on their value stored in the object `.level` (see the example of section 3.1.3).

3.1.2. Structure. The following tables give a preview of the class `CategoryItem`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 3 ▷ Non-importable objects: 0 ▷ Importable methods: 1 ▷ Non-importable methods: 0	

The following table gives a description of the 3 importable objects of the class:

Objects		
Name	Type	Related sections
`.level`	`int`	▷ section 3.1.3
`.source`	`CategoryItem`	▷ section 3.1.3
`.target`	`CategoryItem`	▷ section 3.1.3

Finally, the following table gives a description of the only importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `int` - `list`	- `self`	▷ section 3.1.3

3.1.3. Description of .__init__ (method). The code of the function .__init__ is equipped with the following inputs, among which one can be optional depending on the value of the first argument.

.__init__		
Inputs	Types	Specifications
level	int	necessary
*args	list(CategoryItem)	optional

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If the list args possesses exactly 2 items and args[0].level == level-1 and args[1].level == level-1
Description	In this case, the method allocates the values stored in level, args[0] and args[1] to the objects self.level, self.source and self.target, respectively.

Action 2	
Case	If the list args does not possess exactly 2 items or args[0].level != level-1 or args[1].level != level-1
Description	In this case, the method allocates the values stored in level to the object self.level. The resulting class item is not equipped with the two objects self.source and self.target.

The following example shows how the class can be used as a super class to create functions that behave like functors. First, we create two subclasses of CategoryItem modelling objects and arrows in a certain category.

```
>>> class Obj(CategoryItem):
...     def __init__(self,content):
...       super(Obj, self).__init__(0)
...       self.content = content
...
>>> class Arr(CategoryItem):
...     def __init__(self,A,B,content):
...       super(Arr, self).__init__(1,A,B)
...       self.content = content
...
```

Then, we code a functor taking the previous two classes as the same inputs thanks to their shared ancestry.

```
>>> def functor(cat_item):
...     if cat_item.level == 0:
...       print "object:"
...       print cat_item.content
...       print cat_item.level
...     if cat_item.level == 1:
...       print "arrow:"
...       print cat_item.content
...       print cat_item.level
...       print cat_item.source.content
...       print cat_item.source.level
...       print cat_item.target.content
...       print cat_item.target.level
...
```

Finally, the following lines of code shows that the function `functor` can handle both `Obj` and `Arr` items.

```
>>> a = Obj("a")
>>> b = Obj("b")
>>> f1 = Arr(a,b,"a-->b")
>>> functor(a)
object:
a
0
>>> functor(b)
object:
b
0
>>> functor(f1)
arrow:
a-->b
1
a
0
b
0
```

3.2. Description of usf (class item)

3.2.1. Introduction. The class item `usf` comes from a non-importable class and is equipped with a collection of functions that turned out to be useful during the construction of the present library.

3.2.2. Structure. The following tables give a preview of the class item `usf`. The table given below describes the various dependencies of the class item.

Dependencies	
Class type	Module section
_Useful	N/A
Statistics	
▷ Importable objects: 0	
▷ Non-importable objects: 0	
▷ Importable functions: 4	
▷ Non-importable functions: 0	

The following table gives a description of the 4 importable functions of the class item. The type α shown in the table is a polymorphic type

Functions			
Name	Input types	Output types	Related sections
usf.read_until	- file - list(string) - list(string) - bool	- list(string)	▷ section 3.2.3
usf.fasta	- string	- list(list(string)) - list(string)	▷ section 3.2.4
usf.add_to	- α - list(α)	- bool	▷ section 3.2.5
usf.inclusions	- int - int - int	- gen(list(int))	▷ section 3.2.6

3.2.3. Description of `usf.read_until` (function). The code of the function `usf.read_until` is equipped with the following inputs, among which one is optional.

usf.read_until		
Inputs	Types	Specifications
a_file	file	necessary
separators	list(string)	necessary
EOL_symbols	list(string)	necessary
inclusive = False	bool	optional

The method possesses one action, which we describe below through an example.

Action	
Case	Always
Description	This function reads a file until it reads a character belonging to the input list `EOL_symbols`. It returns the list of pieces of text that were separated by characters belonging to the list `separators`. If the argument `inclusive` is set to `True`, then the last character read in the file is included in the output list. As a result, the output of the function `usf.read_until` is never empty when the argument `inclusive` is set to `True`. On the other hand, if the argument `inclusive` is set to `False`, then the last character read in the file is excluded from the output list.

Let us illustrate the action of the function `usf.read_until` on the following `FASTA` file.

alignment.fa

```
1  >1:A:color_1
2  ACTCGATCTCTG?TCGATCGATCG
3  CCTATCGGATCGATC
4  >1:B:color_1
5  ACTCTAT?TCTATCC-ACTCATCA
6  CCTACTATCTCGAAA
7  >1:C:color_1
8  ACTCTATC----TCCGACT?ATCA
9  CCTACTATCTCGAAA
10 >1:D:color_1
11 ACTATTTA?TTTC----TTTTCTA
12 CCGGGCTGGGGGGGG
```

The following examples illustrate the description that we gave earlier. We start with an example in which the parameter `inclusive` is not specified.

```
>>> with open("alignment.fa","r") as f:
...     keepgoing = True
...     text = list()
...     while keepgoing:
...       line = usf.read_until(f,[" ",":"],["\n"])
...       if line != []:
...         text.append(line)
...       else:
...         keepgoing = False
...     for i in range(len(text)):
...       print text[i]
...
```

Below, we see that setting the parameter `separators` to `[" ",":"]` leads to eliminating these characters from the output.

```
['>1', 'A', 'color_1']
['ACTCGATCTCTG?TCGATCGATCG']
['CCTATCGGATCGATC']
['>1', 'B', 'color_1']
['ACTCTAT?TCTATCC-ACTCATCA']
['CCTACTATCTCGAAA']
['>1', 'C', 'color_1']
['ACTCTATC----TCCGACT?ATCA']
['CCTACTATCTCGAAA']
['>1', 'D', 'color_1']
['ACTATTTA?TTTC----TTTTCTA']
['CCGGGCTGGGGGGGG']
```

We now look at a similar example in which the parameter `inclusive` is set to `True`. Notice that the condition `if line != []` used in the previous piece of code is now `if line != [""]`.

```
>>> with open("alignment.fa","r") as f:
...     keepgoing = True
...     text = list()
...     while keepgoing:
...       line = usf.read_until(f,[" ",":"],["\n"], inclusive = True)
...       if line != [""]:
...         text.append(line)
...       else:
...         keepgoing = False
...     for i in range(len(text)):
...       print text[i]
...
['>1', 'A', 'color_1', '\n']
['ACTCGATCTCTG?TCGATCGATCG', '\n']
['CCTATCGGATCGATC', '\n']
['>1', 'B', 'color_1', '\n']
['ACTCTAT?TCTATCC-ACTCATCA', '\n']
['CCTACTATCTCGAAA', '\n']
['>1', 'C', 'color_1', '\n']
['ACTCTATC----TCCGACT?ATCA', '\n']
['CCTACTATCTCGAAA', '\n']
['>1', 'D', 'color_1', '\n']
['ACTATTTA?TTTC----TTTTCTA', '\n']
['CCGGGCTGGGGGGGG', '\n']
```

3.2.4. Description of `usf.fasta` (function). The code of the function `usf.fasta` is equipped with the following input.

`usf.fasta`		
Inputs	Types	Specifications
`name_of_file`	`string`	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The function takes a `FASTA` file, whose syntax is described in section 3.2.4.1, and returns two lists: ▷ `names`, containing the lists of key words making every sequence labels specified in the file (see section 3.2.4.1); ▷ `sequences`, containing the sequences associated with every sequence label. Furthermore, every list associated with a sequence label in `names` is made of the words that are separated by a colon symbol in the specification of the sequence label.

3.2.4.1. `name_of_file` (variable). The code contained in `name_of_file` should follow a specific syntax that can be described through the following grammar.

$$
\begin{array}{lcl}
\texttt{file} & \to & \texttt{file} \mid \texttt{>}\ \texttt{Label}\ \texttt{\textbackslash n}\ \texttt{Sequence}\ \texttt{file} \mid \varepsilon \\
\texttt{Label} & \to & \texttt{Word}\ \texttt{:}\ \texttt{Label} \mid \texttt{Word} \\
\texttt{Sequence} & \to & \textbf{not}(\texttt{[>]})\ \texttt{Sequence} \mid \varepsilon \\
\texttt{Word} & \to & \textbf{not}(\texttt{[\textbackslash n,:]})\ \texttt{Word} \mid \varepsilon
\end{array}
$$

Because this syntax is compatible with the FASTA format, the file should ideally be specified with the extension .fa or .fasta. For example, the file alignment.fa given in section 3.2.3 constitutes a perfect example of the type of file with which the function usf.fasta should be used.

The following example shows what the output of usf.fasta looks like when applied to the file alignment.fa of section 3.2.3.

```
>>> fa = usf.fasta("alignment.fa")
>>> for i in range(len(fa)):
...     print fa[i]
...
[['1', 'A', 'color_1'], ['1', 'B', 'color_1'], ['1', 'C', 'color_1'], ['1',
'D', 'color_1']]
['ACTCGATCTCTG?TCGATCGATCGCCTATCGGATCGATC', 'ACTCTAT?TCTATCC-ACTCATCACCTACTA
TCTCGAAA', 'ACTCTATC----TCCGACT?ATCACCTACTATCTCGAAA', 'ACTATTTA?TTTC----TTT
TCTACCGGGCTGGGGGGGG']
```

3.2.5. Description of usf.add_to (function). The code of the function usf.add_to is equipped with the following inputs.

usf.add_to		
Inputs	Types	Specifications
element	α	necessary
a_list	list(α)	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The function appends the element element to the list a_list if this element is not present in the list. Otherwise, the element is not appended.

The following examples illustrate the previous description.

```
>>> l = [1, 2, 3, 4, 5]
>>> b = usf.add_to(5,l)
>>> print b, l
False [1, 2, 3, 4, 5]
>>> b = usf.add_to(6,l)
>>> print b, l
True [1, 2, 3, 4, 5, 6]
```

3.2.6. Description of usf.inclusions (generator). The code of the generator usf.inclusions is equipped with the following inputs.

usf.inclusions		
Inputs	Types	Specifications
start	int	necessary
domain	int	necessary
holes	int	necessary

The generator possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The generator outputs the lists `f` whose implicit mappings `i` $\mapsto$ `f[i]` represent increasing inclusions from the ordered set $\{0, 1, ..., \texttt{domain} - \texttt{holes} - 1\}$ to the ordered set $\{\texttt{start}, \texttt{start} + 1, ..., \texttt{start} + \texttt{domain} - 1\}$

The following examples illustrate the previous description. As the reader can notice, the outputs of `usf.inclusions` enumerate the combinations of (`domain` − `holes`) elements among `domain` elements.

```
>>> for i in usf.inclusions(0,5,2):
...     print i
...
[0, 1, 2]
[0, 1, 3]
[0, 1, 4]
[0, 2, 3]
[0, 2, 4]
[0, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 3, 4]
[2, 3, 4]
```

In the following example, we shift the range in which the combinations are computed.

```
>>> for i in usf.inclusions(2,7,3):
...     print i
...
[2, 3, 4, 5]
[2, 3, 4, 6]
[2, 3, 4, 7]
[2, 3, 4, 8]
[2, 3, 5, 6]
[2, 3, 5, 7]
[2, 3, 5, 8]
[2, 3, 6, 7]
[2, 3, 6, 8]
[2, 3, 7, 8]
[2, 4, 5, 6]
[2, 4, 5, 7]
[2, 4, 5, 8]
[2, 4, 6, 7]
[2, 4, 6, 8]
[2, 4, 7, 8]
[2, 5, 6, 7]
[2, 5, 6, 8]
[2, 5, 7, 8]
[2, 6, 7, 8]
[3, 4, 5, 6]
[3, 4, 5, 7]
[3, 4, 5, 8]
[3, 4, 6, 7]
```

```
[3, 4, 6, 8]
[3, 4, 7, 8]
[3, 5, 6, 7]
[3, 5, 6, 8]
[3, 5, 7, 8]
[3, 6, 7, 8]
[4, 5, 6, 7]
[4, 5, 6, 8]
[4, 5, 7, 8]
[4, 6, 7, 8]
[5, 6, 7, 8]
```

3.2.7. Description of usf.join_fibers (function).

3.2.8. Description of usf.parse_string (function).

Presentation of the module `DProgramming.py`

4.1. Description of `Tree` (class)

4.1.1. Introduction. The class `Tree` models the features of a tree structure. A `Tree` item is equipped with an object `.parent` in which it is possible to store information and an object `.children` through which one can specify descendants. A `Tree` item can be constructed recursively from the constructor. `Tree` items whose object `.parent` is equal to the string `"leaf"` are distinguished from the rest of the structure and considered as terminal states. For instance, these terminal states are useful if one wants to enumerate all the paths in the tree. For instance, the class `Tree` is equipped with a method `.paths` that generates all the paths going from the root to a leaf.

4.1.2. Structure. The following tables give a preview of the class `Tree`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 4	
▷ Non-importable objects: 0	
▷ Importable methods: 4	
▷ Non-importable methods: 0	

The following table gives a description of the 4 importable objects of the class:

Objects		
Name	Type	Related sections
`.depth`	`int`	▷ section 4.1.3
`.level`	`int`	▷ section 4.1.3
`.parent`	α	▷ section 4.1.3
`.children`	`list`(`Tree`(α))	▷ section 4.1.3

Finally, the following table gives a description of the 4 importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
.__init__	- α - list(Tree(α))	- self	▷ section 4.1.3
.stdout	- self	- self	▷ section 4.1.4
.levelup	- self	- self	▷ section 4.1.5
.paths	- self	- gen(α)	▷ section 4.1.6

4.1.3. Description of .__init__ (method). The code of the function .__init__ is equipped with the following inputs, among which one can be optional depending on the value of the first argument.

.__init__		
Inputs	Types	Specifications
parent	α	necessary
*args	list(Tree(α))	optional

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If self.parent != "leaf"
Description	In this case, the method allocates the values stored in the input variables parent and args to the objects self.parent and self.children, respectively. The value of the object self.level is set to 0 and the value of the object self.depth is computed as a sum m+1, where m is the maximum value taken among the objects .depth of every Tree item in the list args.

Action 2	
Case	If self.parent == "leaf"
Description	In this case, the method allocates the value stored in the input variable parent to the object self.parent (the object .children is not constructed). The value of the object self.level is set to 0 and the value of the object self.depth is computed as a sum m+1, where m is the maximum value taken among the objects .depth of every Tree item in the list args.

The following example how the constructor can be used to construct a Tree item for which the value contained in the object .depth is 4.

```
>>> leaf = Tree("leaf")
>>> leaf.depth
0
>>> a = Tree("a",[leaf])
>>> a.depth
1
>>> b = Tree("b",[leaf])
>>> c = Tree("c",[leaf])
>>> d = Tree("d",[leaf])
>>> e = Tree("e",[leaf])
>>> ab = Tree("ab",[a,b,leaf])
>>> de = Tree("de",[d,e])
>>> cde = Tree("c(de)",[c,de])
>>> t = Tree("ab(c(de))",[ab,cde])
>>> t.depth
4
>>> t.level
0
>>> t.parent
ab(c(de))
```

4.1.4. Description of `.stdout` (method). The code of the function `.stdout` has no input. The method possesses one action, which we describe below through an example.

Action	
Case	Always
Description	This method displays the tree structure on the standard output where, for every `Tree` item within the recursive structure of the `Tree` item, the line `"["+str(self.level)+"] -> "+str(self.parent)` is displayed through a recursive search. The item `self` is returned as an output.

The following example displays the tree constructed in the example of section 4.1.3.

```
>>> t.stdout()
[0] -> ab(c(de))
.[1] -> ab
..[2] -> a
..[2] -> b
.[1] -> c(de)
..[2] -> c
..[2] -> de
...[3] -> d
...[3] -> e
```

4.1.5. Description of `.levelup` (method). The code of the function `.levelup` has no input. The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	This method recursively increments by 1 all the objects `.level` in the recursive structure of the `Tree` item. The item `self` is returned as an output.

The following example shows the effect of the method `.levelup` on the structure of the tree used in the example of section 4.1.4.

```
>>> t.levelup().stdout()
.[1] -> ab(c(de))
..[2] -> ab
...[3] -> a
...[3] -> b
..[2] -> c(de)
...[3] -> c
...[3] -> de
....[4] -> d
....[4] -> e
>>> t.levelup().stdout()
..[2] -> ab(c(de))
...[3] -> ab
....[4] -> a
....[4] -> b
...[3] -> c(de)
....[4] -> c
....[4] -> de
.....[5] -> d
.....[5] -> e
```

4.1.6. Description of `.paths` (method). The code of the generator `.paths` has no input. The method possesses one action, which we describe below through an example.

Action	
Case	Always
Description	This method generates all the paths going from the root (`self.parent`) to a `Tree` item, belonging the recursive structure of the `Tree` item `self`, whose object `.parent` is equal to the string `"leaf"`.

The following example computes a list containing all the paths of the `Tree` item `t` constructed in section 4.1.3.

```
>>> for i in t.paths()
...     print i
[['ab(c(de))', 'ab', 'a'], ['ab(c(de))', 'ab', 'b'], ['ab(c(de))',
'ab'], ['ab(c(de))', 'c(de)', 'c'], ['ab(c(de))', 'c(de)', 'de', 'd'],
['ab(c(de))', 'c(de)', 'de', 'e']]
```

4.2. Description of `Sequence` (class)

4.2.1. Introduction. The class `Sequence` models the features of a sequence that would belong to a multiple sequence alignment specified in a `FASTA` file (more specifically, see section 3.2.4.1). Its objects `.name` and `.color` contain information that would be specified in the label given before the sequence (according to the `FASTA` format). The sequence itself is stored in the object `.seq`.

4.2.2. Structure. The following tables give a preview of the class `PreOrder`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 3 ▷ Non-importable objects: 0 ▷ Importable methods: 1 ▷ Non-importable methods: 0	

The following table gives a description of the 3 importable objects of the class:

Objects		
Name	Type	Related sections
`.name`	`string`	▷ section 4.2.3
`.seq`	`list(string)`	▷ section 4.2.3
`.color`	`string`	▷ section 4.2.3

Finally, the following table gives a description of the only importable method of the class:

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `string` - `list(string)` - `string`	- `self`	▷ section 4.2.3

4.2.3. Description of `.__init__` (method). The code of the function `.__init__` is equipped with the following inputs.

`.__init__`		
Inputs	Types	Specifications
`name`	`string`	necessary
`sequence`	`list(string)`	necessary
`color`	`string`	necessary

The method possesses one action, which we describe below through an example.

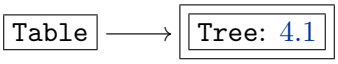
Action	
Case	Always
Description	The method allocates the values of the input variables `name`, `sequence` and `color` to the objects `self.name`, `self.seq`, and `self.color`, respectively.

The following example shows a canonical use of the constructor of the class `Sequence`.

```
>>> sam = Sequence("Samuel",["A","C","T","T","C","g","g","a","T"],"blue")
>>> sam.name
Samuel
>>> sam.  seq
['A','C','T','T','C','g','g','a','T']
>>> sam.color
blue
```

4.3. Description of `Table` (class)

4.3.1. Introduction. The code of the class `Table` uses other functions of the module `DProgramming.py`.



The class `Table` models the features of a dynamic programming score table for the purpose of aligning two pair of sequences of characters. In particular, it encodes the dynamic programming algorithm that allows the type of analysis discussed in [2, Section 4] for the recognition of biological mechanisms.

The class `Table` should be used as shown in the following example. First, use the class `Sequence` (section 4.2) to initialize the table.

```
>>> a = list("abc[defg]hij")
>>> b = list("abc[gfed]hij")
>>> A = Sequence("a",a,"1")
>>> B = Sequence("b",b,"1")
>>> table = Table(A,B)
```

Then, the method `.incidence` should be called to compute matches between the two `Sequence` items given to the constructor of the class `Table`.

```
>>> table.incidence()
>>> table.stdout()
. | . | a | b | c | [ | g | f | e | d | ] | h | i | j |
. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
a | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
b | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
c | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
[ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
d | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
e | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
f | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
g | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
h | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
i | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
j | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
```

Then, the method `.fillout` should be called to fill out the table with the scores used by the dynamic programming algorithm.

```
>>> table.fillout()
>>> table.stdout()
. | . | a | b | c | [ | g | f | e | d | ] | h | i | j |
. | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
a | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
b | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
c | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
[ | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
d | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 |
e | 0 | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
f | 0 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
g | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
```

```
] | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 |
h | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 7 | 7 | 7 |
i | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 7 | 8 | 8 |
j | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 6 | 7 | 8 | 9 |
```

Finally, the method `.dynamic_programming` can be used to store all the valid sequence alignments resulting from the previous scores in a file (here taken to be `"output.fa"`), as shown below. As with the function `open`, the second argument of `.dynamic_programming` takes either `"w"`, to rewrite on the file, or `"a"`, to append the outputs to an existing file.

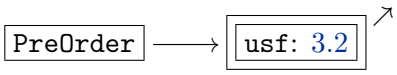
```
>>> table.dynamic_programming("output.fa",option = "w",debug = False,display
= True)
>0:a:1
abc[defg---]hij
>0:b:1
abc[---gfed]hij
>1:a:1
abc[de-fg--]hij
>1:b:1
abc[--gf-ed]hij
>2:a:1
abc[d-efg--]hij
>2:b:1
abc[-g-f-ed]hij
>3:a:1
abc[-defg--]hij
>3:b:1
abc[g--f-ed]hij
>4:a:1
abc[de-f-g-]hij
>4:b:1
abc[--gfe-d]hij
>5:a:1
abc[d-ef-g-]hij
>5:b:1
abc[-g-fe-d]hij
>6:a:1
abc[-def-g-]hij
>6:b:1
abc[g--fe-d]hij
>7:a:1
abc[d--efg-]hij
>7:b:1
abc[-gfe--d]hij
>8:a:1
abc[-d-efg-]hij
>8:b:1
abc[g-fe--d]hij
>9:a:1
abc[--defg-]hij
>9:b:1
abc[gf-e--d]hij
```

```
>10:a:1
abc[de-f--g]hij
>10:b:1
abc[--gfed-]hij
>11:a:1
abc[d-ef--g]hij
>11:b:1
abc[-g-fed-]hij
>12:a:1
abc[-def--g]hij
>12:b:1
abc[g--fed-]hij
>13:a:1
abc[d--ef-g]hij
>13:b:1
abc[-gfe-d-]hij
>14:a:1
abc[-d-ef-g]hij
>14:b:1
abc[g-fe-d-]hij
>15:a:1
abc[--def-g]hij
>15:b:1
abc[gf-e-d-]hij
>16:a:1
abc[d--e-fg]hij
>16:b:1
abc[-gfed--]hij
>17:a:1
abc[-d-e-fg]hij
>17:b:1
abc[g-fed--]hij
>18:a:1
abc[--de-fg]hij
>18:b:1
abc[gf-ed--]hij
>19:a:1
abc[---defg]hij
>19:b:1
abc[gfed---]hij
```

Presentation of the module SegmentCategory.py

5.1. Description of PreOrder (class)

5.1.1. Introduction. The code of the class `PreOrder` uses external modules.



The class `PreOrder` models the features of a pre-ordered set. The pre-order relations are specified through either a file or another `PreOrder` item passed to the constructor. The method `.closure` makes sure that the pre-order axioms are satisfied and will (re-)compute the transitive closure of the pre-order relations stored in the object `.relations` if needed; the method `.geq` returns a Boolean value specifying whether there is a pre-order relation between two given elements of the pre-ordered set; the method `.inf` returns the infimum of two elements of the pre-ordered set; and the method `.presence` returns a Boolean value specifying whether an element belongs to the pre-ordered set.

5.1.2. Structure. The following tables give a preview of the class `PreOrder`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
object	N/A
Statistics	
▷ Importable objects: 4	
▷ Non-importable objects: 0	
▷ Importable methods: 5	
▷ Non-importable methods: 2	

The following table gives a description of the 4 importable objects of the class. The type α shown in the table is a polymorphic type.

Objects		
Name	Type	Related sections
.relations	list(list(α))	▷ section 5.1.3 ▷ section 5.1.4
.transitive	bool	▷ section 5.1.3 ▷ section 5.1.4
.mask	bool	▷ section 5.1.3
.cartesian	int	▷ section 5.1.3 ▷ section 5.1.5 ▷ section 5.1.6

Finally, the following table gives a description of the 5 importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
.__init__	- string - int - list(PreOrder)	- self	▷ section 5.1.3
.closure	- self	- self	▷ section 5.1.4
.geq	- α - α - list	- bool	▷ section 5.1.5
.inf	- α - α - list	- α	▷ section 5.1.6
.presence	- α	- bool	▷ section 5.1.7

5.1.3. Description of .__init__ (method). The code of the function .__init__ is equipped with the following inputs, among which two are optional.

.__init__		
Inputs	Types	Specifications
name_of_file	string	necessary
cartesian = 0	int	optional
*args	list(PreOrder)	optional

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If cartesian > 0 and the list args possesses exactly 1 item
Description	In this case, the list args is assumed to contain a PreOrder item that the method uses to initialize the three objects self.relations, self.transitive, self.mask through componentwise attributions. The object self.cartesian is initialized with the value stored in cartesian.

Action 2	
Case	If cartesian <= 0 or the list args does not possess exactly 1 item
Description	In this case, the method uses the file referenced through the variable name_of_file to fill out the objects of the class. In section 5.1.3.1 (see below), we describe the syntax of the file and how it is parsed.

5.1.3.1. `name_of_file` (variable). The code contained in `name_of_file` should follow a specific syntax that can be described through the following grammar.

```
file      → file | !obj: Obj rel: Rel | obj: Obj rel: Rel | ε
Obj       → Separator Word Comment Obj | ε
Rel       → Text > Text ; Comment Rel | Comment | ε
Separator → not([0-9]+[@]+[A-Z]+[_]+[a-z]) Separator | ε
Word      → [0-9] Word | @ Word | [A-Z] Word | _ Word | [a-z] Word | ε
Comment   → #Text | ε
Text      → Separator Word Text | ε
```

Key words	
`!obj:`/`obj:`	Indicates that the objects of the pre-ordered set are about to be listed. Each object is defined by a `Word` variable in the previous grammar.
`!obj:`	Adds a formal minimal object to the pre-order structure
`rel:`	Specifies that the relations of the pre-ordered set are about to be listed. The relations should only contain variables that were listed after `!obj:` or `obj:`.
`#`	Indicates that the following piece of text is a comment.

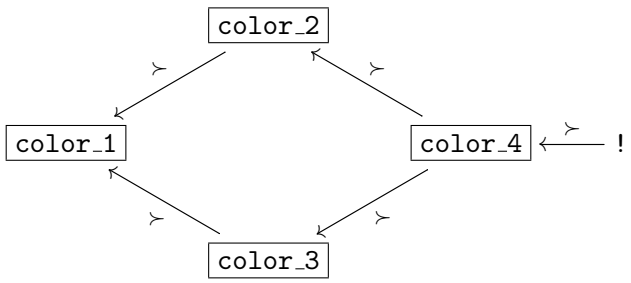
Because this syntax can be compatible with the `YAML` grammar, the file can ideally be specified with the extension `.yml`. Here is an example of such a file.

`preorder.yml`

```
1  #This is an example of a pre-ordered set.
2
3  !obj:#a formal minimal object is added to the pre-order
4  - color_1 #1st object
5  - color_2 #2st object
6  - color_3 #3rd object
7  - color_4 #4th object
8
9  rel:#these are generating relations for the preorder.
10 - color_1 > color_2, color_3;
11 - color_1 > color_3, color_1;
12 - color_3, color_2 > color_4;
```

The previous pre-ordered set can be described by atomic relations shown in diagram (5.1), where the additional object `!` is the formal minimal object added to the pre-order structure.

(5.1)



We now give examples showing how the pre-ordered set `preorder.yml` is used by the constructor of the class `PreOrder`.

After calling the constructor, the object `.relations` will contain the pre-order relations specified in `preorder.yml` as a list of lists (see below). Every internal list starts with a predecessor of all the elements in that list. There is an internal list for every element specified as an object of the pre-ordered set in `preorder.yml`.

```
>>> Omega = PreOrder("preorder.yml")
>>> Omega.relations
[['color_1', 'color_2', 'color_3'], ['color_2', 'color_4'], ['color_3',
'color_4'], ['color_4']]
```

The object `.mask` is set to `True` if a formal mininal object has been added to the structure via the use of `!obj:` and is set to `False` otherwise.

```
>>> Omega.mask
True
```

Since we have not specified the value of the parameter `cartesian`, its value is still 0.

```
>>> Omega.cartesian
0
```

Also, since the method `.closure()` has not been called yet, the object `.transitive` is still equal to `False`.

```
>>> Omega.transitive
False
```

Calling the method `.closure()` will complete the lists inside `.relations` into down-closures for the first element of every list.

```
>>> Omega.closure()
>>> Omega.relations
[['color_1', 'color_2', 'color_3', 'color_4'], ['color_2', 'color_4'],
['color_3', 'color_4'], ['color_4']]
```

We can now verify that the call of the method `.closure()` set the value of the object `.transitive` to `True`.

```
>>> Omega.transitive
True
```

5.1.4. Description of `.closure` (method). The code of the function `.closure` has no input. The method possesses two actions, which we describe below through examples.

Action 1	
Case	If `self.transitive == False`
Description	In this case, the value of the object `self.transitive` is set to `True` and the method completes every list contained in the object `self.relations` into a list that represents the (transitive) down-closure of the first element of each list. The item `self` is returned as an output.

Action 2	
Case	If `self.transitive == True`
Description	No action is taken. The item `self` is returned as an output

Here is an example that uses the file `preorder.yml` of section 5.1.3.1 as an input.

```
>>> Omega = PreOrder("preorder.yml")
>>> Omega.relations
[['color_1', 'color_2', 'color_3'], ['color_2', 'color_4'], ['color_3',
'color_4'], ['color_4']]
>>> Omega.closure().relations
[['color_1', 'color_2', 'color_3', 'color_4'], ['color_2', 'color_4'],
['color_3', 'color_4'], ['color_4']]
```

5.1.5. Description of `.geq` (method). The code of the function `.geq` is equipped with the following inputs.

.geq		
Inputs	Types	Specifications
`element1`	α	necessary
`element2`	α	necessary

The method possesses two actions, which we describe below through examples. Before starting each of these actions, the method calls the method `.closure()` to make sure that the relations contained in the object `.relations` define transitive down-closures.

Action 1	
Case	If `self.cartesian == 0`
Description	The function considers `element1` and `element2` to be elements belonging to the lists contained in `self.relations`. The method first looks for `element1` at the beginning of every list in `self.relations` and if a list is found, the method looks for `element2` in that list.

The following example shows how the method compares element that belong or not to the pre-ordered set.

```
>>> Omega = PreOrder("preorder.yml")
>>> Omega.geq("?","color_1")
False
>>> Omega.geq("color_1","?")
False
>>> Omega.geq("color_1","color_4")
True
>>> Omega.geq("color_4","color_1")
False
>>> Omega.geq("color_2","color_3")
False
```

The second action allows us to simulate any Cartesian product structure on the pre-ordered set.

Action 2	
Case	If `self.cartesian > 0`
Description	The function considers `element1` and `element2` to be lists of elements belonging to the lists contained in `self.relations`. For every element `i` in `range(self.cartesian)`, the method looks for `element1[i]` at the beginning of every list in `self.relations` and if a list is found, it looks for `element2[i]` in that list. Exceptionally for the present case, any component `element2[i]` is allowed to take Boolean values. If `element2[i]` happens to be equal to `True`, then the method skip the search for `element1[i]`.

This second example shows how the method `.geq` extends to the Cartesian product of the pre-order used in the previous example.

```
>>> Cartesian = PreOrder("",2,Omega)
>>> Cartesian.geq(["color_1","color_2"],[Cartesian.mask]*2)
True
>>> Cartesian.geq([Cartesian.mask]*2,["color_1","color_2"])
False
>>> Cartesian.geq(["color_1","color_2"],["color_4","color_4"])
True
>>> Cartesian.geq(["color_1","color_2"],["color_4","color_3"])
False
```

5.1.6. Description of `.inf` (method). The code of the function `.inf` is equipped with the following inputs.

`.inf`		
Inputs	Types	Specifications
`element1`	α	necessary
`element2`	α	necessary

The method possesses two actions, which we describe below through examples. Before starting each of these actions, the method calls the method `.closure()` to make sure that the relations contained in the object `.relations` define transitive down-closures.

Action 1	
Case	If `self.cartesian == 0`
Description	The function considers `element1` and `element2` to be elements belonging to the lists contained in `self.relations`. The method first looks for `element1` and `element2`. If one of them is not found, then `self`.mask is returned. If both elements are present, then the intersection of their down-closures is computed and the first local maxima of the intersection is returned. If the intersection is empty, then `self.mask` is returned instead.

The following example shows how the method compares element that belong or not to the pre-ordered set.

```
>>> Omega = PreOrder("preorder.yml")
>>> Omega.inf("?","color_1")
True
>>> Omega.inf("color_2","color_3")
color_4
```

The second action allows us to simulate any Cartesian product structure on the pre-ordered set.

Action 2	
Case	If `self.cartesian > 0`
Description	The function considers `element1` and `element2` to be lists of elements belonging to the lists contained in `self.relations`. For every element `i` in `range(self.cartesian)`, the method applies Action 1 to the pair (`element1[i]`,`element2[i]`). The output is then appended to a list, which is eventually returned.

This second example shows how the method `.geq` extends to the Cartesian product of the pre-order used in the previous example.

```
>>> Cartesian = PreOrder("",2,Omega)
>>> Cartesian.inf(["color_1","color_2"],["?"]*2)
[True, True]
>>> Cartesian.inf(["color_1","color_2"],["color_4","color_3"])
['color_4', 'color_4']
```

5.1.7. Description of .presence (method). The code of the function .presence is equipped with the following inputs.

.inf		
Inputs	Types	Specifications
element	α	necessary

The method possesses one action, which we describe below through an example.

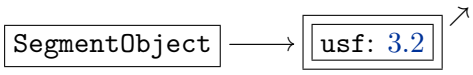
Action	
Case	Always
Description	The function looks for element at the beginning of every list contained in self.relations. If element is found, then True is returned, otherwise the method returns False.

The following examples shows how the method .presence reacts when given various inputs.

```
>>> Omega = PreOrder("preorder.yml")
>>> Omega.presence("?")
False
>>> Omega.presence("color_2")
True
```

5.2. Description of SegmentObject (subclass)

5.2.1. Introduction. The code of the class SegmentObject uses external modules.



The (sub)class SegmentObject models the features of segments (as defined in [2, section 2]). A SegmentObject item can be seen as a tape equipped with a read head, whose position is stored in the object .parse and is displayed through the method .display as a red patch of nodes.

(oooo|ooooooo|ooo|o|oooooo)

The method .patch returns the index of a patch (an area in brackets) containing a node whose position is given as an input. Note that the method .patch starts searching the index associated with the node from where the read head is and only goes in a direction (left or right) specified in the second argument; the method .merge takes a tiling of the segment and merges groups of patches that share the same tiles. The tiling patterns are specified in a list of triples, where each triple contains a start index, a tile length, and an end index; the method .remove removes either a node or a patch (from the segment) at the index given in the first argument: whether a node or a patch is removed is decided upon whether the second argument is equal to "nodes-given" or is not specified.

5.2.2. Structure. The following tables give a preview of the class SegmentObject. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
CategoryItem	section 3
Statistics	
▷ Importable objects: 4 ▷ Non-importable objects: 0 ▷ Importable methods: 5 ▷ Non-importable methods: 1	

The following table gives a description of the 4 importable objects of the class. The type α shown in the table is a polymorphic type.

Objects		
Name	Type	Related sections
.domain	int	▷ section 5.2.3
.topology	list(int * int)	▷ section 5.2.3
.colors	list(list(α))	▷ section 5.2.3
.parse	int	▷ section 5.2.3

Finally, the following table gives a description of the 5 importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
.__init__	- int - list(int * int) - list(list(α))	- self	▷ section 5.2.3
.display	- self	- self	▷ section 5.2.4
.patch	- int - string	- int	▷ section 5.2.5
.merge	- list(int * int * int) - fun: α * α -> α	- SegmentObject(α)	▷ section 5.2.6
.remove	- list - string	- SegmentObject	▷ section 5.2.7

5.2.3. Description of .__init__ (method). The code of the function .__init__ is equipped with the following inputs.

.__init__		
Inputs	Types	Specifications
domain	int	necessary
topology	list(int * int)	necessary
colors	list(α)	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method uses the inputs domain, topology and colors to initialize the objects self.domain, self.topology and self.colors. The object self.parse is given the value 0.

The following examples show how the constructor can be used.

```
>>> t = list()
>>> c = list()
>>> for i in range(5):
...     t = t + [(3*i,3*i+2)]
...     c = c + ["color_1"]
...
>>> s = SegmentObject(20,t,c)
>>> s.domain
20
>>> s.topology
[(0, 2), (3, 5), (6, 8), (9, 11), (12, 14)]
>>> s.colors
['color_1', 'color_1', 'color_1', 'color_1', 'color_1']
>>> s.parse
0
```

5.2.4. Description of `.display` (method). The code of the function `.display` has no input. The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	Displays on the standard output a segment equipped with a 'read head' displayed as a **red** patch of nodes whose index matches the index contained in `self.parse`. The intervals defined by the pairs contained in `self.topology` are displayed as **bold** patches of nodes while the nodes that are not comprised within pairs in `self.topology` are displayed using a normal font. The item `self` is returned as an output.

The following examples show the action of the method `.display` on the segment `s` defined at the end of section 5.2.3.

```
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s.parse = 3
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
```

5.2.5. Description of `.patch` (method). The code of the function `.patch` is equipped with the following inputs, among which one is optional.

`.patch`		
Inputs	Types	Specifications
`position`	`int`	necessary
`search = ">1"`	`string`	optional

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If `search == ">"+str(n)` where `n` is of the type `int`
Description	The method returns the index of a pair `(x,y)` in `self.topology` if the value `position` is between `x` and `y`. The list `self.topology` is parsed from the pair at index `self.parse` to the pair at index `len(self.topology)-1` with an incrementation equal to `n`. If the value `position` is within the range of one of the parsed pairs in `self.topology`, then the index of the pair (within `self.topology`) is returned. In any other case, the value -1 is returned.

Action 2	
Case	If `search == "<"+str(n)` where `n` is of the type `int`
Description	The method returns the index of a pair `(x,y)` in `self.topology` if the value `position` is between `x` and `y`. The list `self.topology` is parsed from the pair at index `self.parse` to the pair at index `0` with an incrementation equal to `n`. If the value `position` is within the range of one of the parsed pairs in `self.topology`, then the index of the pair (within `self.topology`) is returned. In any other case, the value -1 is returned.

The following examples show the action of the method `.patch` on the segment `s` considered at the end of section 5.2.4.

```
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s.patch(5,"<1")
1
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s.patch(7,">1")
2
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s.patch(11,">2")
-1
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s.patch(4,"<2")
-1
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
```

5.2.6. Description of `.merge` (method). The code of the function `.merge` is equipped with the following inputs.

`.merge`		
Inputs	Types	Specifications
`folding_format`	`list(int * int * int)`	necessary
`infimum`	`fun: α * α -> α`	necessary

The method possesses one action.

Action	
Case	Always
Description	The method returns a new segment that is a merged version of `self`. Specifically, the patches of `self` are merged according to the tiling patterns specified in `folding_format`, in which a tiling pattern is given by a pair `(x,l,y)`. For every pair `(x,l,y)` in `folding_format`, the method will transitively merge any subset of patches whose indices in `self.topology` belong to a same interval of the form `(x+k*l,min(x+(k+1)*l,y))`. For each merging, the resulting pair that is included in the new topology consists of the first component of the first merged pair and the last component of the last merged pair. The color associated with that pair is the output of the recursive application `f(_,...(_,f(_,f(_,_)))` of the function `f` := `infimum`.

The following examples will probably make the previous description more explicit. We use the `preOrder` constructed from the file `preorder.yml` shown in section 5.1.3.1.

```
>>> Omega = PreOrder("preorder.yml")
>>> t = list()
>>> c = list()
>>> for i in range(5):
...     t = t + [(3*i,3*i+2)]
...     if i % 2 == 0:
...       c = c + ["color_2"]
...     if i % 2 == 1:
...       c = c + ["color_3"]
...
>>> s = SegmentObject(20,t,c)
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.merge([(0,1,3)],Omega.inf)
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1.colors
['color_2', 'color_3', 'color_2', 'color_3', 'color_2']
>>> s1 = s.merge([(0,2,3)],Omega.inf)
>>> s1.display()
(oooooo|oooooo|ooo|ooooo)
>>> s1.colors
['color_4', 'color_4', 'color_2']
>>> s1 = s.merge([(0,3,3)],Omega.inf)
>>> s1.display()
(ooooooooo|ooo|ooo|ooooo)
>>> s1.colors
['color_4', 'color_3', 'color_2']
>>> s1 = s.merge([(0,4,3)],Omega.inf)
>>> s1.display()
(oooooooooooo|ooo|ooooo)
>>> s1.colors
['color_4', 'color_2']
>>> s1 = s.merge([(0,5,3)],Omega.inf)
```

```
>>> s1.display()
(oooooooooooo|ooo|ooooo)
>>> s1.colors
['color_4', 'color_2']
```

5.2.7. Description of .remove (method). The code of the function .remove is equipped with the following inputs, among which one is optional.

.__init__		
Inputs	Types	Specifications
a_list	list	necessary
option = "patches-given"	string	optional

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If option != "nodes-given"
Description	In this case, the method removes every patch whose index is an integer i in a_list.

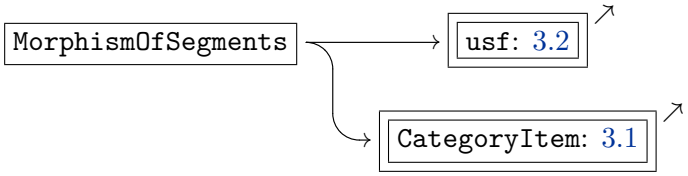
Action 2	
Case	If option == "nodes-given"
Description	In this case, the method removes every patch whose index is an integer i for which there exists an index j in a_list such that i = self.patch(j).

The following examples shows the effects of choosing either two options. We use the segment s defined at the end of section 5.2.6.

```
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([0],"patches-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([0],"nodes-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([1],"patches-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([1],"nodes-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([2],"patches-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([2],"nodes-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> s1 = s.remove([0,1,2,6],"patches-given")
>>> s1.display()
(oooooooo|ooo|ooo|ooooo)
>>> s1 = s.remove([0,1,2,6],"nodes-given")
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
```

5.3. Description of `MorphismOfSegments` (subclass)

5.3.1. Introduction. The code of the class `MorphismOfSegments` uses external modules.



The (sub)class `MorphismOfSegments` models the features of a morphism of segments (as defined in [2]). As such, its constructor takes a source segment, a target segment, a list `f1` of increasing integers describing the image of an injection `i` $\mapsto$ `f1[i]` relating the domain of the source segment to the domain of the target segment, and a pre-order relation compatible with the pre-ordered sets of two the segments.

5.3.2. Structure. The following tables give a preview of the class `MorphismOfSegments`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`CategoryItem`	section 3
Statistics	
▷ Importable objects: 3	
▷ Non-importable objects: 0	
▷ Importable methods: 1	
▷ Non-importable methods: 1	

The following table gives a description of the 3 importable objects of the class:

Objects		
Name	Type	Related sections
`.defined`	`bool`	▷ section 5.3.3
`.f0`	`list(int)`	▷ section 5.3.3
`.f1`	`list(int) or string`	▷ section 5.3.3

Finally, the following table gives a description of the only importable method of the class. The type α shown in the table is a polymorphic type.

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `SegmentObject(`α`)` - `SegmentObject(`α`)` - `list(int)` - `fun: `α` * `α` -> bool`	- `self`	▷ section 5.3.3

5.3.3. Description of `.__init__` (method). The code of the function `.__init__` is equipped with the following inputs.

`.__init__`		
Inputs	Types	Specifications
`source`	`SegmentObject(`α`)`	necessary
`target`	`SegmentObject(`α`)`	necessary
`f1`	`list(int) or string`	necessary
`geq`	`fun: `α` * `α` -> bool`	necessary

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If `f1 == "id"`
Description	In this case, the super objects `self.source` and `self.target` are given the value of the variables `source` and `target`, respectively. The object `self.f1` is given the value `"id"` and the object `self.f0` will given the list containing the integers `self.target.patch(i)`, for every index `i` in `range(self.source.topology)` if `self.f0` defines a function. In this case, the object `self.defined` is set to `True`.

Action 2	
Case	If `f1 != "id"` and `f1` is of the type `list(int)`
Description	In this case, the super objects `self.source` and `self.target` are given the value of the variables `source` and `target`, respectively. The object `self.f1` is given the list stored in `f1` and the object `self.f0` will be given the list containing the integers `self.target.patch(f1[i])`, for every index `i` in `range(self.source.topology)`, if `self.f0` defines a function. In this case, the object `self.defined` is set to `True`.

The following examples illustrate the previous actions. We consider the lists `t` and `c` defined in section 5.2.6.

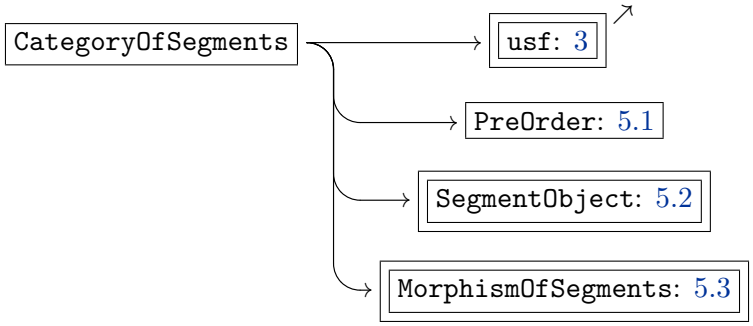
```
>>> s = SegmentObject(20,t,c)
>>> t = SegmentObject(25,t,c)
>>> s.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> t.display()
(ooo|ooo|ooo|ooo|ooo|oooooooooo)
>>> s1 = s.remove([2])
>>> s1.display()
(ooo|ooo|ooo|ooo|ooo|ooooo)
>>> t1 = t.remove([0,2,3])
>>> t1.display()
(ooo|ooo|oooooo|ooo|ooooo)
>>> f1 = range(20)
>>> m = MorphismOfSegments(s1,t1,f1,Omega.geq)
>>> m.defined
True
>>> m.f0
[-1, 0, -1, 1]
>>> m.f1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Not all morphisms are well-defined. For instance, the following examples contained non-well-defined morphisms of segments. We obtained these examples by changing the topology or the mapping `f1`. As can be seen, the resulting object `.f0` is an empty list.

```
>>> s2 = s1.merge([(1,2,2)],Omega.inf)
>>> s2.display()
(ooo|ooooooooo|ooo|ooooo)
>>> m = MorphismOfSegments(s2,t1,f1,Omega.geq)
>>> m.defined
False
>>> m.f0
[]
>>> f1 = f1[1:]+[20]
>>> f1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> m = MorphismOfSegments(s1,t1,f1,Omega.geq)
>>> m.defined
False
>>> m.f0
[]
```

5.4. Description of CategoryOfSegments (class)

5.4.1. Introduction. The code of the class `CategoryOfSegments` uses other functions of the module `SegmentCategory.py` as well as external modules.



The class `CategoryOfSegments` models the features of a category of segments. The class is initialized by passing a `PreOrder` item to its constructor and allows one to know or compute information related to a category structure. The method `.identity` returns a Boolean value that specificies whether there may exist an identity morphism between two `SegmentObject` items (these may be saved in different places in the memory); the method `.initial` returns an local initial object in the category, where the local aspect is determined by the colors of the segment; the method `.homset` computes the hom-set of a pair of `SegmentObject` items.

5.4.2. Structure. The following tables give a preview of the class `CategoryOfSegments`. The table given below describe the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
object	N/A
Statistics	
▷ Importable objects: 1	
▷ Non-importable objects: 0	
▷ Importable methods: 4	
▷ Non-importable methods: 0	

The following table gives a description of the only importable object of the class:

Objects		
Name	Type	Related sections
.preorder	PreOrder	▷ section 5.4.3

Finally, the following table gives a description of the 4 importable methods of the class. The type α shown in the table is a polymorphic type.

Methods			
Name	Input types	Output types	Related sections
.__init__	- PreOrder	- self	▷ section 5.4.3
.identity	- SegmentObject - SegmentObject	- bool	▷ section 5.4.4
.initial	- int - list(list(α))	- SegmentObject	▷ section 5.4.5
.homset	- SegmentObject - SegmentObject	- gen(MorphismOfSegments)	▷ section 5.4.6

5.4.3. Description of .__init__ (method). The code of the function .__init__ is equipped with the following inputs.

.__init__		
Inputs	Types	Specifications
preorder	PreOrder	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method allocates the value of the variable preorder to the object self.preorder

The following example shows how the constructor of the class CategoryOfSegments can be used with the class PreOrder (see section 5.1). We use the file preorder.yml displayed in section 5.1.3.1.

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> Seg.preorder.relations
[['color_1', 'color_2', 'color_3'], ['color_2', 'color_4'], ['color_3',
'color_4'], ['color_4']]
>>> Seg.preorder.mask
True
```

5.4.4. Description of .identity (method). The code of the function .identity is equipped with the following inputs.

.identity		
Inputs	Types	Specifications
segment1	SegmentObject	necessary
segment2	SegmentObject	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method tests whether each type of objects .domain, topology and colors are equal for both inputs segment1 and segment2.

In the following example, we use the file `preorder.yml` displayed in section 5.1.3.1 and the method `.initial` detailed in section 5.4.5.

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> s = Seg.initial(20,"color_1")
>>> s.display()
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
>>> t = Seg.initial(20,"color_1")
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
>>> t.display()
>>> Seg.identity(s,t)
True
```

If we now use the method `.remove`, detailed in section 5.2.7, to change the colors of `t`, then the identity does not hold anymore, as sown in the following example.

```
>>> t = t.remove([15])
>>> t.display()
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
>>> Seg.identity(s,t)
False
```

5.4.5. Description of `.initial` (method). The code of the function `.initial` is equipped with the following inputs.

`.initial`		
Inputs	Types	Specifications
`domain`	`int`	necessary
`color`	α	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method returns the `SegmentObject` item whose object `.topology` consists of the pairs `(i,i)` for every `i` in `range(domain)` and whose object `.colors` is equal to the list `[color]*domain`.

The following example shows what a `SegmentObject` item generated by the method `.initial` looks like. We use the file `preorder.yml` displayed in section 5.1.3.1 and the constructor of the lass `PreOrder` (see section 5.1).

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> s = Seg.initial(20,[1,0])
>>> s.display()
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
>>> s.colors
[[1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0],
[1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1, 0], [1,
0], [1, 0]]
```

5.4.6. Description of `.homset` (method). The code of the generator `.homset` is equipped with the following inputs

.homset		
Inputs	Types	Specifications
source	SegmentObject	necessary
target	SegmentObject	necessary

The method possesses one action, which we describe below through examples. We use the file preorder.yml displayed in section 5.1.3.1 and the method .initial detailed in section 5.4.5.

Action	
Case	Always
Description	The method outputs all MorphismOfSegments items for which the object .source is equal to source and the object .target is equal to target relative to the preorder structure contained in self.preorder.

The following example shows what the outputs of the method .homset look like. We use the file preorder.yml displayed in section 5.1.3.1, the method .initial detailed in section 5.4.5, the method .merge detailed in section 5.2.6 and the constructor of the class PreOrder (see section 5.1).

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> s = Seg.initial(8,"color_1")
>>> s = s.merge([(0,2,7)],Omega.inf)
>>> t = Seg.initial(12,"color_1")
>>> t = t.merge([(0,3,10)],Omega.inf)
>>> s.display()
(oo|oo|oo|oo)
>>> t.display()
(ooo|ooo|ooo|oo|o)
>>> for m in Seg.homset(s,t):
...     print m.f1
...
[0, 1, 3, 4, 6, 7, 9, 10]
[0, 1, 3, 4, 6, 8, 9, 10]
[0, 1, 3, 4, 7, 8, 9, 10]
[0, 1, 3, 5, 6, 7, 9, 10]
[0, 1, 3, 5, 6, 8, 9, 10]
[0, 1, 3, 5, 7, 8, 9, 10]
[0, 1, 4, 5, 6, 7, 9, 10]
[0, 1, 4, 5, 6, 8, 9, 10]
[0, 1, 4, 5, 7, 8, 9, 10]
[0, 2, 3, 4, 6, 7, 9, 10]
[0, 2, 3, 4, 6, 8, 9, 10]
[0, 2, 3, 4, 7, 8, 9, 10]
[0, 2, 3, 5, 6, 7, 9, 10]
[0, 2, 3, 5, 6, 8, 9, 10]
[0, 2, 3, 5, 7, 8, 9, 10]
[0, 2, 4, 5, 6, 7, 9, 10]
[0, 2, 4, 5, 6, 8, 9, 10]
[0, 2, 4, 5, 7, 8, 9, 10]
[1, 2, 3, 4, 6, 7, 9, 10]
[1, 2, 3, 4, 6, 8, 9, 10]
[1, 2, 3, 4, 7, 8, 9, 10]
[1, 2, 3, 5, 6, 7, 9, 10]
[1, 2, 3, 5, 6, 8, 9, 10]
[1, 2, 3, 5, 7, 8, 9, 10]
[1, 2, 4, 5, 6, 7, 9, 10]
[1, 2, 4, 5, 6, 8, 9, 10]
[1, 2, 4, 5, 7, 8, 9, 10]
```

Presentation of the module `AlignedFunctor.py`

6.1. Description of `PointedSet` (class)

6.1.1. Introduction. The class `PointedSet` models the features of a pointed set. It is made of a list `.symbols`, representing the underlying set, and an integer `.index` indicating where the point is locted whithin `.symbols`.

6.1.2. Structure. The following tables give a preview of the class `PreOrder`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 2	
▷ Non-importable objects: 0	
▷ Importable methods: 2	
▷ Non-importable methods: 0	

The following table gives a description of the 2 importable objects of the class. The type α shown in the table is a polymorphic type.

Objects		
Name	Type	Related sections
`.symbols`	`list`(α)	▷ section 6.1.3
`.index`	`int`	▷ section 6.1.3

Finally, the following table gives a description of the 2 importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `list`(α) - `int`	- `self`	▷ section 6.1.3
`.point`	- `self`	- α	▷ section 6.1.3

6.1.3. Description of `.__init__` (method). The code of the function `.__init__` is equipped with the following inputs.

.__init__		
Inputs	Types	Specifications
symbols	string	necessary
index	int	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method allocates the value stored in the input variable symbols to the object self.symbols. If the index is in the range of the list self.symbols, then the value of the input variable index is passed to the object self.index. Otherwise, the object self.index is set to 0.

In the following example, we see how the object .index can be used with the object .symbols to return the 'point' of the PointedSet item.

```
>>> E = PointedSet(["-","A","C","G","T"],0)
>>> E.symbols
['-','A','C','G','T']
>>> E.index
0
>>> E.symbols[E.index]
-
```

6.1.4. Description of .point (method). The code of the function .point has no input. The method possesses one action, which we describe below through an example.

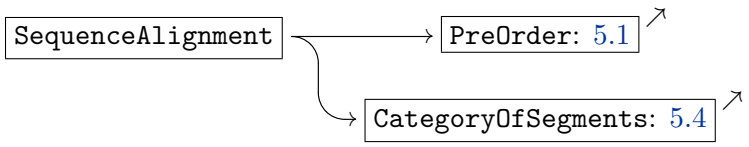
Action	
Case	Always
Description	The method returns the point self.symbols[self.index] of the structure.

The following example ilustrates the use of the method .point.

```
>>> E = PointedSet(["A","C","G","T","epsilon"],4)
>>> E.point()
epsilon
```

6.2. Description of SequenceAlignment (class)

6.2.1. Introduction. The code of the class SequenceAlignment uses external modules.



The class SequenceAlignment models the features of a sequence alignment functor, as defined in [2]. The images of the sequence alignment functor are stored in the object .database and can be queried throught the method .eval. The method also computes the images of the right Kan extension of this functor through the method .ran (TO BE CODED). The objects of the extending category (see [2, Definition 3.25]), which is used to compute this right Kan extension, can be obtained through the method .extending_category.

6.2.2. Structure. The following tables give a preview of the class `SequenceAlignment`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 5 ▷ Non-importable objects: 0 ▷ Importable methods: 4 ▷ Non-importable methods: 0	

The following table gives a description of the 5 importable objects of the class. The type α shown in the table is a polymorphic type.

Objects		
Name	Type	Related sections
`.env`	`Environment`	▷ section 6.2.3
`.Seg`	`list`	▷ section 6.2.3
`.indiv`	`list`	▷ section 6.2.3
`.base`	`list(SegmentObject)`	▷ section 6.2.3
`.database`	`list`(α)	▷ section 6.2.3

Finally, the following table gives a description of the 4 importable methods of the class:

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `Environment` - `CategoryOfSegments` - `list(SegmentObject)` - `list`(α)	- `self`	▷ section 6.2.3
`.eval`	- `SegmentObject`	- α or `list`	▷ section 6.2.4
`.extending_category`	- `SegmentObject`	- `gen(int*` `MorphismOfSegments)`	▷ section 6.2.5
`.ran`	- `CategoryItem`	- α or `list`	▷ section 6.2.6

6.2.3. Description of `.__init__` (method). The code of the function `.__init__` is equipped with the following inputs.

`.__init__`		
Inputs	Types	Specifications
`env`	`Environment`	necessary
`indiv`	`list`	necessary
`base`	`list(SegmentObject)`	necessary
`database`	`list`(α)	necessary

The method possesses one action, which we describe below through an example.

Action	
Case	Always
Description	The method allocates the values stored in the input variables `env`, `indiv`, `base` and `database` to the objects `self.env`, `self.indiv`, `self.base` and `self.database`. It also uses the object `self.env.spec` to compute the (`self.env.spec`)-fold Cartesian product of the pre-ordered set encoded by `self.env.Seg.preorder` and allocates to the object `self.Seg` the `CategoryOfSegments` item that can be constructed from it.

The following examples show how to use the constructor of the class in combination with the classes `PreOrder` (see section 5.1), `CategoryOfSegments` (see section 5.4), `PointedSet` (see section 6.1) and `Environment` (see section 6.3). We use the file `preorder.yml` displayed in section 5.1.3.1.

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> E = PointedSet(["-","A","C","G","T"],0)
>>> Env = Environment(Seg,E,2,["color_2"]*2)
>>> base = [Env.Seg.initial(8,["color_1"]*2), Env.Seg.initial(8,["color_2"]*2),
Env.Seg.initial(5,["color_2"]*2), Env.Seg.initial(7,["color_3"]*2)]
```

In practice, the following list should contain actual sequence alignments, namely each `string` item should be replaced with a succession of `string` items representing the sequences of a sequence alignment.

```
>>> database = [[["Alignment:1:1"], ["Alignment:1:2"]], [["Alignment:2:1"],
["Alignment:2:2"]], [["Alignment:3:1"],["Alignment:3:2"],
["Alignment:3:3"]], [["Alignment:4:1"],["Alignment:4:2"]]]
```

The following code shows how to recover the data passed to the constructor using the various objects of the class.

```
>>> Seqali = SequenceAlignment(Env,["A","B","C","D"],base,database)
>>> Seqali.indiv
['A', 'B', 'C', 'D']
>>> for i in range(len(Seqali.base)):
...     print str(i)+") color:  "+ str(Seqali.base[i].colors[
Seqali.base[i].parse ])
...     Seqali.base[i].display()
...     for j in range(len(Seqali.database[i])):
...       for k in range(len(Seqali.database[i][j])):
...         print Seqali.database[i][j][k]
...     print ""
...
0) color:  ['color_1', 'color_1']
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

1) color:  ['color_2', 'color_2']
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

2) color:  ['color_2', 'color_2']
```

```
(o|o|o|o|o)
Alignment:3:1
Alignment:3:2
Alignment:1:3

3) color:  ['color_3', 'color_3']
(o|o|o|o|o|o|o)
Alignment:4:1
Alignment:4:2
```

6.2.4. Description of .eval (method). The code of the function .eval is equipped with the following input.

.eval		
Inputs	Types	Specifications
segment	SegmentObject	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method returns the image of the sequence alignment functor for the given input SegmentObject item. If the input does not belong to the list stored in self.base, then the empty list is returned.

The following example shows that the images of the SegmentObject items stored in the object .base recovers the elements stored in the object .database. We use the Environment item Env and the SequenceAlignment item Seqali constructed in section 6.2.3.

```
>>> for i in range(len(Seqali.base)):
...     print "image for base[" + str(i) +"]"
...     Seqali.base[i].display()
...     sal = Seqali.eval(Seqali.base[i])
...     for j in range(len(sal)):
...       for k in range(len(sal[j])):
...         print sal[j][k]
...       print ""
...
image for base[0]
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

image for base[1]
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2
```

```
image for base[2]
(o|o|o|o|o)
Alignment:3:1
Alignment:3:2
Alignment:3:3

image for base[3]
(o|o|o|o|o|o|o)
Alignment:4:1
Alignment:4:2
```

For every SegmentObject item that is not in the object .base, the method .eval will return the empty list, as shown below.

```
>>> s = Env.Seg.initial(7,["color_1"]*2)
>>> s.display()
(o|o|o|o|o|o|o)
>>> sal = Seqali.eval(s)
>>> for j in range(len(sal)):
...     for k in range(len(sal[j])):
...       print sal[j][k]
...     print ""
...
>>>
```

6.2.5. Description of .extending_category (method). The code of the generator .extending_category is equipped with the following input.

.extending_category		
Inputs	Types	Specifications
segment	SegmentObject	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method generates the set of objects of the *extending category* defined in [2, Definition 3.25]. More specifically, the method returns the list of pairs (i,m) such that m is a MorphismOfSegments item whose source object is segment and whose target object is self.base[i].

The following example shows an example of calculation of an extending category.

```
>>> diag = Seqali.extending_category(Env.Seg.initial(7,["color_1"]*2))
>>> for i,m in diag:
...     print i
...     print m.f1
...     print m.f0
...     m.source.display()
...     m.target.display()
...     for j in range(len(Seqali.database[i])):
...       for k in range(len(Seqali.database[i][j])):
...         print Seqali.database[i][j][k]
...       print ""
...
```

```
0
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[0, 1, 2, 3, 4, 5, 7]
[0, 1, 2, 3, 4, 5, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[0, 1, 2, 3, 4, 6, 7]
[0, 1, 2, 3, 4, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[0, 1, 2, 3, 5, 6, 7]
[0, 1, 2, 3, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[0, 1, 2, 4, 5, 6, 7]
[0, 1, 2, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[0, 1, 3, 4, 5, 6, 7]
[0, 1, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2
```

```
0
[0, 2, 3, 4, 5, 6, 7]
[0, 2, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

0
[1, 2, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o|o)
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:1:1
Alignment:1:2

1
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[0, 1, 2, 3, 4, 5, 7]
[0, 1, 2, 3, 4, 5, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[0, 1, 2, 3, 4, 6, 7]
[0, 1, 2, 3, 4, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[0, 1, 2, 3, 5, 6, 7]
[0, 1, 2, 3, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2
```

```
1
[0, 1, 2, 4, 5, 6, 7]
[0, 1, 2, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[0, 1, 3, 4, 5, 6, 7]
[0, 1, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[0, 2, 3, 4, 5, 6, 7]
[0, 2, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

1
[1, 2, 3, 4, 5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o|o)
Alignment:2:1
Alignment:2:2

3
id
[0, 1, 2, 3, 4, 5, 6]
(o|o|o|o|o|o|o)
(o|o|o|o|o|o|o)
Alignment:4:1
Alignment:4:2
```

6.2.6. Description of `.ran` (method). The code of the function `.ran` is equipped with the following input.

`.ran`		
Inputs	Types	Specifications
`cat_item`	`CategoryItem`	necessary
`B = ""`	`string`	optional

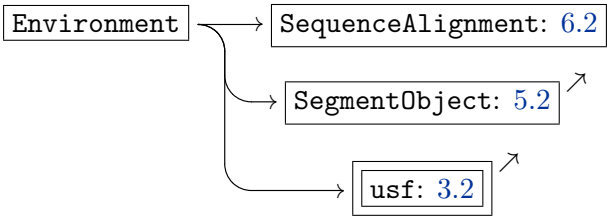
The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method computes the images of the right Kan extension of the functor encoded by the method `self.eval`.

TO BE CODED

6.3. Description of `Environment` (class)

6.3.1. Introduction. The code of the class `Environment` uses other functions of the module `AlignedFunctor.py` as well as external modules.



The class `Environment` models the features of an aligned environment functor, as defined in [2, section 3]. As with aligned environment functors, this structure is associated with a `PointedSet` item `.pset`. The class is also equipped with a fiber operation (pullback along a point in the image of the aligned environment functor)

$$\begin{array}{ccc} \{\tau\} & \overset{\subseteq}{-\!\!=\!\!\to} & \mathbf{Seg}(\Omega) \\ {\scriptstyle !}\downarrow & & \downarrow{\scriptstyle \mathbf{A}E_b^{\varepsilon}} \\ \mathbf{1} & \xrightarrow[\text{input}]{} & \mathbf{Set} \end{array}$$

and a sequence alignment functor constructor call. Specifically, the method `.segment` returns a segment that is the pullback of the aligned environment functor above any input list that represents an element in one of its images. If the input list contains a character that is not in the object `.pset.symbols`, then the node associated with that character is masked in the returned segment. Finally, the class `Environment` is equipped with a method `.seqali` that constructs a sequence aligment functor from a file of sequence alignments, as shown in the discussion of [2, Example 3.22].

6.3.2. Structure. The following tables give a preview of the class `Environment`. The table given below describes the various dependencies of the class.

Dependencies	
Superclass ancestry	Module section
`object`	N/A
Statistics	
▷ Importable objects: 4 ▷ Non-importable objects: 0 ▷ Importable methods: 3 ▷ Non-importable methods: 0	

The following table gives a description of the 4 importable objects of the class. The type α shown in the table is a polymorphic type.

Objects		
Name	Type	Related sections
`.Seg`	`CategoryOfSegments`(α)	▷ section 6.3.3
`.pset`	`PointedSet`(α)	▷ section 6.3.3
`.spec`	`int`	▷ section 6.3.3
`.b`	`list`(α)	▷ section 6.3.3

Finally, the following table gives a description of the 3 importable methods of the class. The type β shown in the table is a polymorphic type.

Methods			
Name	Input types	Output types	Related sections
`.__init__`	- `CategoryOfSegments`(α) - `PointedSet`(α) - `int` - `list`(α)	- `self`	▷ section 6.3.3
`.segment`	- `list`(β) - α	- `SegmentObject`	▷ section 6.3.4
`.seqali`	- `string`	- `SequenceAlignment`	▷ section 6.3.5

6.3.3. Description of `.__init__` (method). The code of the function `.__init__` is equipped with the following inputs.

`.__init__`		
Inputs	Types	Specifications
`Seg`	`CategoryOfSegments`(α)	necessary
`pset`	`PointedSet`(α)	necessary
`exponent`	`int`	necessary
`threshold`	`list`(α)	necessary

The method possesses two actions, which we describe below through examples.

Action 1	
Case	If $\forall$ `i` such that `self.Seg.preorder.presence(self.b[i]) == True`
Description	In this case, the method allocates the values stored in the input variables `Seg`, `pset`, `exponent`, and `threshold` to the objects `self.Seg`, `self.pset`, `self.exponent`, and `self.threshold`, respectively. In addition, the method checks whether the length of the list `self.b` is equal to `self.spec`. If not, then `self.b` is completed into such a list by appending copies of the value `self.Seg.preorder.mask` to it.

Action 2	
Case	If $\exists$ `i` such that `self.Seg.preorder.presence(self.b[i]) == False`
Description	In this case, the method allocates the values stored in the input variables `Seg`, `pset`, `exponent`, and `threshold` to the objects `self.Seg`, `self.pset`, `self.exponent`, and `self.threshold`, respectively, and replaces every element `self.b[i]` for which the case stated above holds with the Boolean value `self.Seg.preorder.mask`. If the length of the list `self.b` is not equal to `self.spec`, then `self.b` is completed into such a list by appending copies of the value `self.Seg.preorder.mask` to it.

The following examples show how to use the constructor of the class in combination with the classes `PreOrder` (see section 5.1), `CategoryOfSegments` (see section 5.4), and `PointedSet` (see section 6.1). We use the file `preorder.yml` displayed in section 5.1.3.1.

```
>>> Omega = PreOrder("preorder.yml")
>>> Seg = CategoryOfSegments(Omega)
>>> E = PointedSet(["-","A","C","G","T"],0)
```

We start by illustrating a case in which Action 1 is applied by the method.

```
>>> Env = Environment(Seg,E,4,["color_2"]*4)
>>> Env.Seg.preorder.relations
[['color_1', 'color_2', 'color_3'], ['color_2', 'color_4'], ['color_3',
'color_4'], ['color_4']]
>>> Env.pset.symbols
['-', 'A', 'C', 'G', 'T']
>>> Env.pset.point()
-
>>> Env.spec
4
>>> Env.b
['color_2', 'color_2', 'color_2', 'color_2']
```

The following example illustrates a case in which the object `.b` does not contain valid elements and whose length is too short.

```
>>> Env = Environment(Seg,E,4,["color_10","color_2"])
>>> Env.Seg.preorder.relations
[['color_1', 'color_2', 'color_3'], ['color_2', 'color_4'], ['color_3',
'color_4'], ['color_4']]
>>> Env.pset.symbols
['-', 'A', 'C', 'G', 'T']
>>> Env.pset.point()
-
>>> Env.spec
4
>>> Env.b
[True, 'color_2', True, True]
```

6.3.4. Description of `.segment` (method). The code of the function `.segment` is equipped with the following inputs.

`.__init__`		
Inputs	Types	Specifications
`a_list`	`list`(β)	necessary
`color`	α	necessary

The method possesses one action, which we describe below through an example.

Action	
Case	Always
Description	The method takes a list of symbols `a_list` whose elements are assumed to belong to the object `self.Env.pset.symbols` and returns the pull-back (a `SegmentObject` item) of the underlying environment functor (defined by `self`) above the element represented by `a_list`. Every element of the list `a_list` that is not a symbol in `self.Env.pset.symbols` is removed from the topology of the `SegmentObject` item.

The following example shows what the effect of the method `.segment` on a list of strings looks like. We use the `Environment` item defined at the end of section 6.3.3.

```
>>> r = Env.segment(["A","T","G","ERROR","-","-","G","?","?"],"color_1")
>>> r.display()
(o|o|o|o|o|o|o|oo)
>>> r.colors
['color_1', 'color_1', 'color_1', 'color_1', 'color_1', 'color_1']
```

6.3.5. Description of `.seqali` (method). The code of the function `.seqali` is equipped with the following input.

`.seqali`		
Inputs	Types	Specifications
`name_of_file`	`string`	necessary

The method possesses one action, which we describe below through examples.

Action	
Case	Always
Description	The method copies the construction given [2, Example 3.22] and produces a sequence aligment functor (*i.e.* a `SequenceAlignment` item – see section 6.2) from a `FASTA` file of sequence alignments. The input file, referenced through the input variable `name_of_file`, should follow the format specified in section 3.2.4.1. More specifically, every sequence label succeeding the usual key symbol `>` should take the form `group_label:individual:color`, where `group_label` indicates a reference number gathering sequences under a same sequence alignment, where `individual` is the name of the individual from which the sequence comes, and `color` is the color that will be used to color the segment associated with the sequence.

Let us consider the following `FASTA` file of sequence alignments whose sequence labels takes the form `group_label:individual:color`.

alignment.fa

```
>1:A:color_1
ACTCGATCTCTG?TCGATCGATCG
CCTATCGGATCGATC
>1:B:color_1
ACTCTAT?TCTATCC-ACTCATCA
CCTACTATCTCGAAA
>1:C:color_1
ACTCTATC----TCCGACT?ATCA
CCTACTATCTCGAAA
>1:D:color_1
ACTATTTA?TTTC----TTTTCTA
CCGGGCTGGGGGGGG
>2:A:color_1
ACTCGATCGGATC-AT??CGATCG
CCTATCGGATCGATCG
>2:B:color_1
ACTCC?CCTA--CTGGGCTGCTCA
CCTACTATCTCGAAAG
>2:C:color_1
ACTCTCTATC?CTCTATC---TCA
CCTACTATC?CGAAAG
>2:D:color_1
ACTATT-??T--AC--CTTTTCTA
CCGGGCTTTTCGAAAA
```

The following example shows what the effect of the applying the method `.seqali` on the previous file is. In particular, we display the outputs of the method `.eval` applied on the `SegmentObject` items contained in the object `.base`.

```
>>> Seqali = Env.seqali("alignment.fa")
>>> for i in range(len(Seqali.base)):
...     print "image for base[" + str(i) + "]"
...     Seqali.base[i].display()
...     print "color:  "+str(Seqali.base[i].colors[Seqali.base[i].parse])
...     sal = Seqali.eval(Seqali.base[i])
...     for j in range(len(sal)):
...       for k in range(len(sal[j])):
...         print sal[j][k]
...       print ""
...
image for base[0]
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o)
color:  ['color_1', 'color_1', 'color_1', 'color_1']
ACTCGATCTCTG?TCGATCGATCGCCTATCGGATCGATC
ACTCTAT?TCTATCC-ACTCATCACCTACTATCTCGAAA
ACTCTATC----TCCGACT?ATCACCTACTATCTCGAAA
ACTATTTA?TTTC----TTTTCTACCGGGCTGGGGGGGG
```

```
image for base[1]
(o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o|o
|o|o|o|o)
color:  ['color_1', 'color_1', 'color_1', 'color_1']
ACTCGATCGGATC-AT??CGATCGCCTATCGGATCGATCG
ACTCC?CCTA--CTGGGCTGCTCACCTACTATCTCGAAAG
ACTCTCTATC?CTCTATC---TCACCTACTATC?CGAAAG
ACTATT-??T--AC--CTTTTCTACCGGGCTTTTCGAAAA
```

Presentation of the module PartitionCategory.py

7.1. Description of _image_of_partition

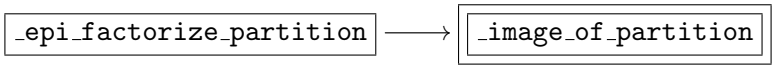
The function _image_of_partition takes a list of elements and returns the list of its elements without repetition in the order in which they can be accessed from the left to the right.

```
1 def _image_of_partition(partition):
2   """ the source code of this function can be found in iop.py """
3   return the_image
```

This corresponds to returning the image object of the underlying partition of the list.

```
>>> print(_image_of_partition([3,3,2,1,1,2,4,5,6,5,2,6]))
[3, 2, 1, 4, 5, 6]
>>> print(_image_of_partition(['A',4,'C','C','G',4,0,0,1,'a','A']))
['A', 4, 'C', 'G', 0, 1, 'a']
```

7.2. Description of _epi_factorize_partition



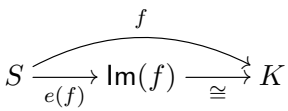
The function _epi_factorize_partition relabels the elements of a list with non-negative integers. It starts with the integer 0 and allocates a new label by increasing the previously allocated label by 1. The first element of the list always receives the label 0 and the highest integer used in the relabeling equals the length of the image (section 7.1) of the list decreased by 1.

```
1 def _epi_factorize_partition(partition):
2   """ the source code of this function can be found in efp.py """
3   return epimorphism
```

Even though a list already encodes an epimorphism, the goal of the function

_epi_factorize_partition

is to return a canonical *choice* of epimorphism.

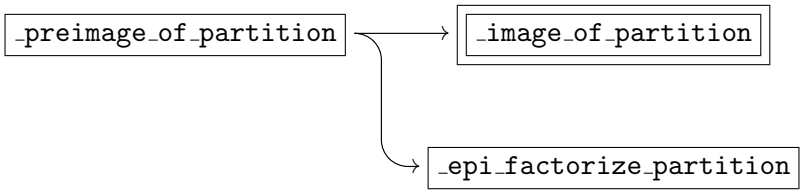


This choice ensures that two partitions characterized by the same set of universal properties are *equal* as python lists. In [4, Appendix A], this type of construction is formulated in terms of a factorization system in the category of sets (see the diagram above).

```
>>> p = [3,3,2,1,1,2,4,5,6,5,2,6]
>>> print(_epi_factorize_partition(p))
[0, 0, 1, 2, 2, 1, 3, 4, 5, 4, 1, 5]
>>> im = _image_of_partition(p)
>>> print(_epi_factorize_partition(im))
[0, 1, 2, 3, 4, 5]
>>> print(_epi_factorize_partition(['A',4,'C','C','G',4,0,0,1,'a','A']))
[0, 1, 2, 2, 3, 1, 4, 4, 5, 6, 0]
```

Note that the function does not literally relabel the input list, but allocates a new space in the memory to store the relabeled list.

7.3. Description of _preimage_of_partition



The function `_preimage_of_partition` takes a list and returns the list of the lists of indices that index the same element.

```
1 def _preimage_of_partition(partition):
2   """ the source code of this function can be found in piop.py """
3   return the_preimage
```

From the point of view of partitions [4, Appendix A], the returned list `the_preimage` (see the code above) is the preimage of the underlying epimorphism $f : S \to K$ of the input `partition`, where the preimage of f is defined as the K-indexed set of the fibers of the epimorphism.

$$\mathsf{PreIm}(f) = \{f^{-1}(k)\}_{k\in K}$$

Note that, from an implementation viewpoint, the set K might not be equipped with an obvious order relation, which makes it difficult to define the preimage of $f : S \to K$ as a python list. To rectify this flaw, the preimage is computed with respect to the canonical epimorphism $e(f) : S \to \mathsf{Im}(f)$ whose codomain is equipped with the natural order on integers (see section 7.2).

$$\mathsf{Preim}(f) := \{e(f)^{-1}(k)\}_{k\in \mathsf{Im}(f)}$$

```
>>> p = ['a','a',2,2,3,3,'a']
>>> print(_preimage_of_partition(p))
[[0, 1, 6], [2, 3], [4, 5]]
>>> print(_epi_factorize_partition(p))
[0, 0, 1, 1, 2, 2, 0]
```

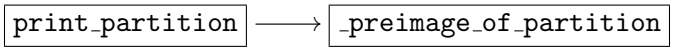
```
>>> p = [2,1,0,6,5,4,2,1,0]
>>> print(_preimage_of_partition(p))
[[0, 6], [1, 7], [2, 8], [3], [4], [5]]
>>> print(_epi_factorize_partition(p))
[0, 1, 2, 3, 4, 5, 0, 1, 2]
```

In the first example given above:

- the list `[0,1,6]` is the fiber of the element `'a'` and its index in the preimage is 0;
- the list `[2,3]` is the fiber of the element `2` and its index in the preimage is 1;
- the list `[4,5]` is the fiber of the element `3` and its index in the preimage is 2.

The preimage will always order its fibers with respect to the order in which the elements of the input list appear.

7.4. Description of print_partition

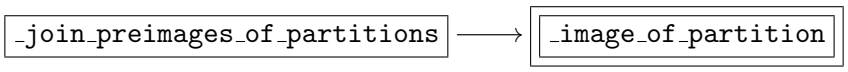


The function `print_partition` is a debug function that takes a list of elements and prints its preimage on the standard output.

```
1 def print_partition(partition):
2   print(_preimage_of_partition(partition))
```

See section 7.3 for examples.

7.5. Description of _join_preimages_of_partitions



The function `_join_preimages_of_partitions` takes two lists of lists of indices (the indices can be repeated and should only be non-negative integers) as well as a Boolean value and returns the list of the maximal unions of internal lists that intersect within the concatenation of the two input lists (see the examples below).

```
1 def _join_preimages_of_partitions(preimage1,preimage2,speed_mode):
2   """ the source code of this function can be found in jpop.py """
3   return the_join
```

While the two input lists `preimage1` and `preimage2` could be two outputs of the procedure

`_preimage_of_partition(−)`

for two input lists of the same length, the Boolean value `speed_mode` would indicate whether one of the two input lists may contain at least two different sublists with the same index, as shown below.

`[[0, 1], [0, 4], [2, 3, 4, 5]]`

Note that the global variable `FAST` is reserved to this use.

```
>>> print(FAST)
True
```

From the point of view of partitions, the composition of `_preimage_of_partition` with `_join_preimages_of_partitions` would amount to computing the coproduct of two partitions. Since a category of partitions is also a partially ordered set, this coproduct is also the *join* of the two partitions, which explains the name of the procedure.

For illustration, if we consider the following two lists of lists of indices

```
>>> p1 = [[0, 3], [1, 4], [2]]
>>> p2 = [[0, 1], [2], [3], [4]]
```

we can notice that

- the internal list [0,3] of p1 intersects with the internal lists [0,1] and [3] in p2;
- the internal list [0,1] of p1 intersects with the internal lists [1,4] and [1] in p2;
- the internal list [1,4] of p1 intersects with the internal list [4] in p2;

and

- the internal list [2] of p1 only intersects with the internal list [2] in p2,

so that we have

```
>>> print(_join_preimages_of_partitions(p1,p2,FAST))
[[1, 4, 0, 3], [2]]
```

In terms of implementation, the program

(7.1) `_join_preimages_of_partitions(p1,p2,FAST)`

considers each internal list of p1 and searches for the lists of p2 that intersect it. If an intersection is found between two internal lists, it merges the two internal lists in p1 and empties that of p2 (the list is emptied and *not* removed in order to preserve a coherent indexing of the elements of p2). The function continues until all the possible intersections have been checked.

Here is a detail of what program (7.1) does with respect to the earlier example:

The element 0 of [0,3] is searched in the list [0,1] of p2;

The element 0 is found;

The lists [0,3] and [0,1] are merged in p1 and [0,1] is emptied from p2 as follows:

```
p1 = [[0, 3, 1], [1, 4], [2]]
p2 = [[], [2], [3], [4]]
```

Because the element 0 has now been found in p2 and the third input was set to FAST, no other sublist of p2 is supposed to contain the element 0 and the search of the element 0 stops here. Note that if not(FAST) were given in the third argument, then the earlier union operation would also be operated on the remaining sublists of p2.

The element 3 of [0, 3] is searched in the list [] of p2;

The element 3 is not found (continues);

The element 3 of [0, 3] is searched in the list [2] of p2;

The element 3 is not found (continues);

The element 3 of [0, 3] is searched in the list [3] of p2;

The element 3 is found;

The lists [0,3] and [3] are merged in p1 and [3] is emptied from p2 as follows:

```
p1 = [[0, 3, 1], [1, 4], [2]]
p2 = [[], [2], [], [4]]
```

The element 3 has now been found in p2 and does not need to be searched again.

All elements of the initial list [0, 3] have been searched.

The first lists of p1 is appended to p2 in order to ensure the transitive computation of the maximal unions through the next interations.

The list [0, 3, 1] of p1 is emptied as follows:

```
p1 = [[], [1, 4], [2]]
p2 = [[], [2], [], [4], [0, 3, 1]]
```

Repeat the previous procedure with respect to the list [1, 4] of p1. We obtain the following pair:

```
p1 = [[], [], [2]]
p2 = [[], [2], [], [], [], [1, 4, 0, 3]]
```

Repeat the previous procedure with respect to the remaining list [2] of p1. We obtain the following pair:

```
p1 = [[], [], []]
p2 = [[], [], [], [], [], [1, 4, 0, 3], [2]]
```

The function stops because there is no more list to process in p1. The output is all the non-empty lists of p2; i.e. [[1, 4, 0, 3], [2]]

Note that, because of the iterative nature of the previous algorithm, the procedure

_join_preimages_of_partitions(−,−,−)

does not necessarily presents its output in the same way as the procedure

_preimage_of_partition(−)

does. For instance, while the index 0 will always be contained in the first list of the output of _preimage_of_partition, it might not be contained in the first list of the output of _join_preimages_of_partitions as illustrated below.

```
>>> l = _preimage_of_partition([1,2,4,5,1,2,3,2,2,1,3])
>>> print(l)
[[0, 4, 9], [1, 5, 7, 8], [2], [3], [6, 10]]
>>> m = _preimage_of_partition([1,2,5,4,2,2,5,6,5,7,8])
>>> print(m)
[[0], [1, 4, 5], [2, 6, 8], [3], [7], [9], [10]]
>>> print(_join_preimages_of_partitions(l,m))
[[3], [6, 10, 2, 1, 5, 7, 8, 0, 4, 9]]
```

Interestingly, the procedure _join_preimages_of_partitions can also be used to compute the intersection-free closure of a set of sets, as shown below.

```
>>> a = [[1,0,2,1,3,2,5,4,6],[15,15,18,0,13],[7,11,12,22],[23,12]]
>>> closure_of_a = _join_preimages_of_partitions(a,a,not(FAST))
>>> print(closure_of_a)
[[15, 18, 0, 13, 1, 2, 3, 5, 4, 6], [23, 12, 7, 11, 22]]
```

7.6. Description of EquivalenceRelation (class)

EquivalenceRelation ⟶ _join_preimages_of_partitions

The class EquivalenceRelation possesses two objects, namely

- .classes (list of lists);
- .range (integer);

and three methods, namely

- .__init__ (constructor);
- .closure;
- .quotient.

The constructor .__init__ takes between 1 and 2 arguments: the first argument should either be an empty list or a list of lists of indices (i.e. non-negative integers) and the second argument, which is optional when the first argument is not an empty list, should be an integer that is greater than or equal to the maximum index contained in the first input, if it exists.

```
 1 class EquivalenceRelation:
 2   #The objects of the class are:
 3   #.classes (list of lists);
 4   #.range (integer);
 5   #The following constructor takes between 1 and 2 arguments,
 6   #the first one being a list and the second being an integer.
 7   def __init__(self,*args):
 8   """ the source code of this constructor can be found in cl_er.py """
 9   def closure(self):
10     self.classes = _join_preimages_of_partitions(self.classes,
   self.classes,not(FAST))
11   def quotient(self):
12   """ the source code of this function can be found in cl_er.py """
13     return the_quotient
```

If the first input is not empty, then it is stored in the object .classes while the object .range receives:

- either the second input, when this second input is given;
- or the maximum index contained in the first input when no second input is given.

```
>>> eq1 = EquivalenceRelation([[0,1,2,9],[7,3,8,6],[4,9,5]])
>>> print(eq1.classes)
[[0, 1, 2, 9], [7, 3, 8, 6], [4, 9, 5]]
>>> print(eq1.range)
9
>>> eq2 = EquivalenceRelation([[0,1,2,9],[7,3,8,7],[9,15]],18)
>>> print(eq2.classes)
[[0, 1, 2, 9], [7, 3, 8, 7], [9, 15]]
>>> print(eq2.range)
18
```

If the first input is empty, then the second argument is required. In this case, the object .classes receive the lists containing all the singleton lists containing the integers from 0 to the integer given in the second argument, which is stored in the object .range.

```
>>> eq3 = EquivalenceRelation([],5)
>>> print(eq3.classes)
[[0], [1], [2], [3], [4], [5]]
```

The method .closure() replaces the content of the object .classes with the transitive closure of its classes. After this procedure, the object .classes describes an actual equivalence relation (modulo the singleton equivalence classes, which do not need to be specified for obvious reasons).

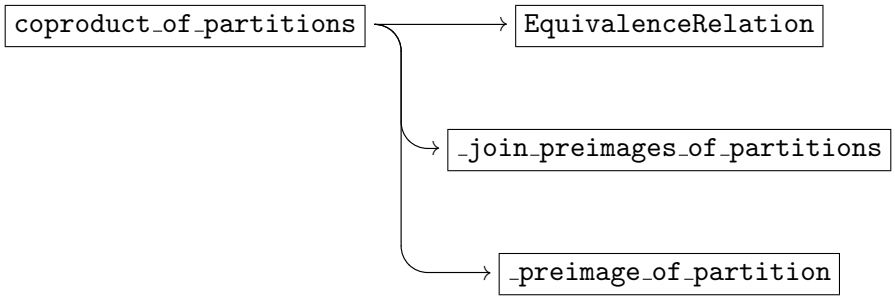
```
>>> eq1.closure()
>>> print(eq1.classes)
[[7, 3, 8, 6], [4, 9, 5, 0, 1, 2]
```

```
>>> eq2.closure()
>>> print(eq2.classes)
[[7, 3, 8], [9, 15, 0, 1, 2]]
```

The method `.quotient()` returns a list of integers whose length is equal to the integer contained in the object `.range` decreased by 1 and whose non-trivial fibers are those contained in the object `.classes` after a call of the method `.closure()`.

```
>>> print(eq1.quotient())
[1, 1, 1, 0, 1, 1, 0, 0, 0, 1]
>>> print(eq2.quotient())
[1, 1, 1, 0, 2, 3, 4, 0, 0, 1, 5, 6, 7, 8, 9, 1, 10, 11, 12]
```

7.7. Description of coproduct_of_partitions



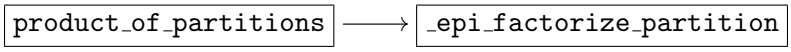
The function `coproduct_of_partitions` takes two lists of the same legnth and returns their coproduct (or join) as partitions. Specifically, the procedure outputs the quotient of the join of their preimages (see the piece of code given below). If the two input lists do not have the same length, then an error message is outputted and the program is aborted.

```
 1 def coproduct_of_partitions(partition1,partition2):
 2   if len(partition1) == len(partition2):
 3     #Returns the coproduct of two partitions as the quotient of the
 4     #equivalence relation induced by the join of the preimages
 5     #of the two partitions.
 6     the_join = EquivalenceRelation(_join_preimages_of_partitions(
   _preimage_of_partition(partition1),_preimage_of_partition(partition2),
   FAST))
 7     return the_join.quotient()
 8   else:
 9     print("Error:  in coproduct_of_partitions:  lengths do not match.")
10     exit()
```

Note that the outputs of the procedure `coproduct_of_partitions` do not necessarily belong to the set of outputs of the procedure `_epi_factorize_partition`. The reason comes from the way in which the procedure `_join_preimages_of_partitions` is implemented (see the end of section 7.5).

```
>>> l = [1,2,4,5,1,2,3,2,2,1,3]
>>> m = [1,2,5,4,2,2,5,6,5,7,8]
>>> c = coproduct_of_partitions(l,m)
>>> print(c)
[1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1]
>>> print(_epi_factorize_partition(c))
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

7.8. Description of product_of_partitions



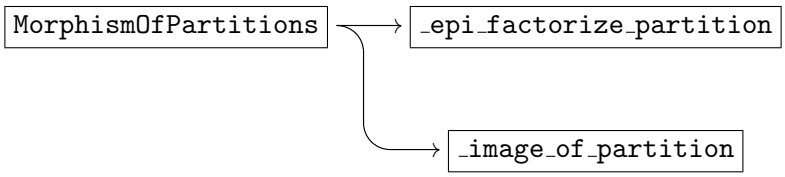
The function product_of_partitions takes two lists and returns a list that is the relabeling of the zipping of the two lists (i.e. the list of pairs of elements with corresponding indices in each of the input lists) via the procedure _epi_factorize_partition.

```
 1 def product_of_partitions(partition1,partition2):
 2   #The following line checks if the product of the two lists is possible.
 3   if len(partition1) == len(partition2):
 4     #Constructs the list of pairs of element with the
 5     #same index in the two lists, and then relabels
 6     #the pairs using _epi_factorize_partition.
 7     return _epi_factorize_partition(zip(partition1,partition2))
 8   else:
 9     print("Error:  in product_of_partitions:  lengths do not match.")
10     exit()
```

The function outputs an error if the two input lists do not have the same length.

```
>>> product_of_partitions([1,1,1,1,2,3],['a','b','c','c','c','c'])
[0, 1, 2, 2, 3, 4]
>>> product_of_partitions([1,1,1,1,2],['a','b','c','c','c','c'])
Error:  in product_of_partitions:  lengths do not match.
```

7.9. Description of MorphismOfPartitions (class)



The class MorphismOfPartitions possesses three objects, namely

- .arrow (list)
- .source (list)
- .target (list)

and a constructor .__init__. The consructor .__init__ takes two lists as well as an optional argument and stores, in the object .arrow, the list that describes, if it exists, the (unique) morphism of partitions from the first input list (seen as a partition) to the second input list (seen as a partition). If the morphism does not exist, then the method returns an error message unless the value False was given as a third input.

The canonical epimorphisms associated with the partitions of the first and second input lists (see section 7.2) are stored in the objects .source and .target, respectively.

```
1 class MorphismOfPartitions:
2   #The objects of the class are:
3   #.arrow (list);
4   #.source (list);
5   #.target (list).
6   def __init__(self,source,target,*args):
7   """ the source code of this constructor can be found in cl_mop.py """
```

The list that is contained in the object `.arrow` is the image of the function `source` $\mapsto$ `target`(`source`) that can be constructed from the parametrization $t \mapsto$ `source`$[t]$ and $t \mapsto$ `target`$[t]$, which are given by the list structure of the two input lists `target` and `source`. This is illustrated below in more detail.

For illustration, let us consider the following pair of lists.

```
>>> p1 = [0,1,2,3,3,4,5]
>>> p2 = [0,1,2,3,3,3,1]
```

To construct the function `source` $\mapsto$ `target`(`source`), we can first try to construct the graph (`source`$[t]$, `target`$[t]$), for which we use the procedure `zip`.

```
>>> p3 = zip(p1,p2)
>>> print(p3)
[(0, 0), (1, 1), (2, 2), (3, 3), (3, 3), (4, 3), (5, 1)]
```

The image of the zipping is then as follows:

```
>>> p4 = _image_of_partition(p3)
>>> print(p4)
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 1)]
```

We can see that for each pair `(x,y)` in `p4`, every component `x` is mapped to a unique image `y` so that `p4` defines the *graph* of the morphism of partitions between `p1` and `p2`. The constructor `.__init__` then records, in the object `.arrow`, the second projections of the pairs contained in `p4`, which also corresponds to the image of the underlying graph encoded by `p4`.

```
>>> m = MorphismOfPartitions(p1,p2)
>>> print(m.arrow)
[0, 1, 2, 3, 3, 1]
```

If there exists no morphism from the first input list to the second input list, then the function outputs an error message.

For example, if we modify `p2` as follows

```
>>> p2 = [0,1,2,3,6,3,1]
```

then the image of the zipping of `p1` and `p2` is as follows:

```
>>> p4 = _image_of_partition(zip(p1,p2))
>>> print(p4)
[(0, 0), (1, 1), (2, 2), (3, 3), (3, 6), (4, 3), (5, 1)]
```

As can be seen, the argument `3` is 'mapped' to two different images, namely `3` and `6`. In this case, the constructor `.__init__` exits the program with an error message.

Here is a complete example summarizing the previous explanation.

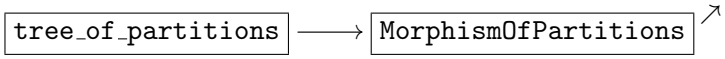
```
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,4,4,2])
>>> print(m.source)
[0, 1, 2, 3, 3, 4, 5]
>>> print(m.target)
[0, 1, 2, 3, 3, 3, 1]
>>> print(m.arrow)
[0, 1, 2, 3, 3, 1]
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2])
Error:  in MorphismOfPartitions.__init__:  source and target are not
compatible.
>>> m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2],False)
```

Finally, note that the constructor of `MorphismOfPartitions` can be used to test whether there is a morphism between two partitions by using the key words `try` and `except` and setting the thrid argument to `False`, as illustrated below.

```
>>> try:
>>>   m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,4,4,2],False)
>>>   print("True")
>>> except:
>>>   print("False")
True
>>> try:
>>>   m = MorphismOfPartitions([1,6,5,3,3,4,2],[1,2,5,4,6,4,2],False)
>>>   print("True")
>>> except:
>>>   print("False")
False
```

Presentation of the module AsciiTree.py

8.1. Description of tree_of_partitions



The function `tree_of_partitions` takes a list of lists whose pairs of successive lists can be related via `MorphismOfPartitions` items (see section 7.9)

$$[\overbrace{\underbrace{l_1,l_2}_{\text{pair of succ. lists}},l_3}^{\text{pair of succ. lists}},\ldots,l_n]$$

and returns the actual lists of `MorphismOfPartitions` items between these.

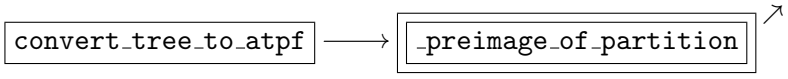
```
1 def tree_of_partitions(partitions):
2   """ the source code of this function can be found in top.py """
3   return the_tree
```

The input list should always start with the target of the first arrow, then present its source, which should also be the target of the next arrow, etc.

```
>>> l = [[0,0,0,0,0,0], [0,0,0,0,0,1], [0,0,2,2,2,1], [0,0,3,1,1,2],
    [0,1,2,3,4,5]]
>>> tree = tree_of_partitions(l)
>>> for i in range(len(tree)):
...   print(tree[len(tree)-1-i].source)
...   print("  |\n  | "+"arrow num "+str(len(tree)-i) + ":  " + str(
    tree[len(tree)-1-i].arrow) + "\n  |\n  V")
... print(tree[0].target)
[0, 1, 2, 3, 4, 5]
  |
  | arrow num 4:  [0, 0, 1, 2, 2, 3]
  |
  V
```

```
[0, 0, 1, 2, 2, 3]
  |
  | arrow num 3:  [0, 1, 1, 2]
  |
  V
[0, 0, 1, 1, 1, 2]
  |
  | arrow num 2:  [0, 0, 1]
  |
  V
[0, 0, 0, 0, 0, 1]
  |
  | arrow num 1:  [0, 0]
  |
  V
[0, 0, 0, 0, 0, 0]
```

8.2. Description of convert_tree_to_atpf

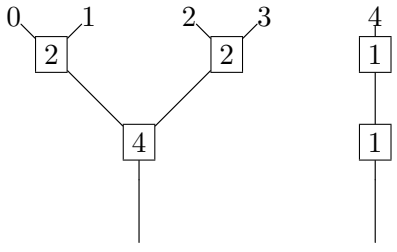


The function convert_tree_to_atpf takes a list of MorphismOfPartitions item (as returned by the procedure tree_of_partitions) and converts it into its associated *ascii tree pre-format* (abbrev. atpf), which is a regular expression of type ATPF given by the following double grammar rules.

The grammar terms	and their associated weights
$\texttt{ATPF} := [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k];$	
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Tree}_1, \texttt{Tree}_2, \ldots, \texttt{Tree}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Tree}_i);$
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Leaf}_1, \texttt{Leaf}_2, \ldots, \texttt{Leaf}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Leaf}_i);$
$\texttt{Leaf} := (\texttt{weight}(\texttt{Leaf}), l),$ where l is a list	$\texttt{weight}(\texttt{Leaf}) := \texttt{len}(l);$

The idea of such a construction is to give access to the number of leaves contained in each fork of a tree. This type of information will later be used to display trees with ascii characters on the console. For illustration, the following line gives the example of an atpf that describes the forest displayed below it.

```
atpf = [(4, [(2, [0, 1]), (2, [2, 3])]), (1, [(1, [4])])]
```



Note that the number of levels in the trees can be linked to what could be called the *depth* of the bracketing structure of the atpf. Specifically, we define the *depth* of an afpt according

to the following recursive equations, relative to the definition of the terms given above.

$$\begin{aligned}
&\texttt{depth}(\texttt{ATPF}) := \max\{\texttt{depth}(\texttt{Tree}_i) \mid i = 1, \ldots, k\};\\
&\texttt{depth}(\texttt{Tree}) := \max\{\texttt{depth}(\texttt{Tree}_i) \mid i = 1, \ldots, k\} + 1;\\
&\texttt{depth}(\texttt{Tree}) := \max\{\texttt{depth}(\texttt{Leaf}_i) \mid i = 1, \ldots, k\} + 1;\\
&\texttt{depth}(\texttt{Leaf}) := 1;
\end{aligned}$$

The procedure `convert_tree_to_atpf` then returns a list and an integer, where the list is the atpf of the input (i.e. of the tree) and the integer is equal to the depth of the atpf, which is equal to `len(tree)+1`.

```
1 def _convert_tree_to_atpf(tree):
2   """ the source code of this function can be found in ctta.py """
3   return (the_atpf,len(tree)+1)
```

Since the depth is only returned for parsing purposes (see section 8.3 and section 8.4), the main task of the procedure `convert_tree_to_atpf` is to compute the atpf associated with the input list of composable `MorphismOfPartitions` items by considering the successive preimagesof the objects `.arrow` of each `MorphismOfPartitions` item contained in the list.

For illustration, consider the following list of partitions:

```
>>> a = [0,1,0,0,0,0]
>>> b = [0,2,0,0,0,1]
>>> c = [0,4,2,3,3,5]
```

This list can be associated with an obvious sequence of `MorphismOfPartitions` item that is constructed by the procedure `tree_of_partitions` (see section 8.1). Below, we make the steps of this construction explicit for the list `[a, b ,c]`.

First, one wants to consider the preimage of the last partition `c`. This preimage is outputted by the function `_preimage_of_partition` (see section 7.3) as shown below.

```
>>> _preimage_of_partition(c)
[[0], [1], [2], [3, 4], [5]]
```

The internal lists appearing in the previous list are important for our next step. More specifically, we want to follow the recursive definition of the grammar of atpfs (given above) and take the lists `[0]`, `[1, 2]`, `[3, 4]`, and `[5]` to be the initial values of the recursion so that the first level of the atpf is as shown below, where the red numbers are the weight of the `Leaf` terms (see grammar rules above).

```
the_atpf = [(1, [0]), (1, [1]]), (1, [2]]), (2, [3, 4]), (1, [5])]
```

For the next level, we need to compute the preimage of the object `.arrow` encoding the morphism `c` $\to$ `b`. Recall that the list contained in this object encodes the mapping rules associated with the morphism `c` $\to$ `b`. Below, we display this mapping up to relabeling of the lists `c` and `b` as `[0,1,2,3,3,4]` and `[0,1,0,0,0,2]`, respectively.

```
>>> f = MorphismsOfPartitions(c,b)
>>> for i in range(len(f.arrow)):
...   print("f:  "+str(i)+" |--> "+str(f.arrow[i]))
f:  0 |-> 0
f:  1 |-> 1
f:  2 |-> 0
f:  3 |-> 0
f:  4 |-> 2
```

Now, since `b` has three elements in its image, the `MorphismOfPartitions` item `f` possesses three fibers, which are given by the following list of lists.

```
>>> fiber = _preimage_of_partition(f.arrow)
>>> print(fiber)
[[0, 2, 3], [1], [4]]
```

Following the atpf grammar and, more specifically, the syntax for the `Tree` terms, we now want to use the list `fiber` computed above to create the next level of the atpf. The idea is to replace the intergers living in `fiber` with the elements of the list `the_atpf`. Precisely, we want to replace:

```
fiber[0][0] = 0 with the_atpf[0]
fiber[0][1] = 2 with the_atpf[2]
fiber[0][2] = 3 with the_atpf[3]
fiber[1][0] = 1 with the_atpf[1]
fiber[2][0] = 4 with the_atpf[4]
```

Doing so, the list `fiber` is turned into the following list.

```
fiber = [[(1, [0]), (1, [2]]), (2, [3, 4])], [(1, [1]])], [(1, [5])]]
```

To complete the construction of the next level of the atpf, there remains to compute the weight for each internal list. Precisely, we can see that

- the weight of `[(1, [0]), (1, [2]]), (2, [3, 4])]` is 1+1+2 = 4;
- the weight of `[(1, [1]])]` is 1;
- the weight of `[(1, [5])]]` is 1.

We then equip each list with its weight by using tuples, as shown below.

```
the_atpf = [(4,[(1, [0]), (1, [2]]), (2, [3, 4])]), (1,[(1, [1]]))], (1,[(1,
[5])])]
```

We then repeat the previous procedure with, this time, the fiber of the morphism `b` $\to$ `a` and the earlier list `the_atpf` so that the final atpf is of the following form.

```
the_atpf = [(5, [(4, [(1, [0]), (1, [2]), (2, [3, 4])]), (1, [(1, [5])])]),
(1, [(1, [(1, [1])])])]
```

The user that verify that the previous list conrresponds to the output of the following command.

```
>>> print(convert_tree_to_atpf(tree_of_partitions([a,b,c]))[0])
```

8.3. Description of `convert_atpf_to_atf`

The function `convert_atpf_to_atf` takes an atpf and its depth (see section 8.2) and returns the associated *ascii tree format* (abbrev. atf), which is a modified version of an atpf in which one substracts all the weights by the rightmost weight of the next level, as shown by the following grammar rules

$$\begin{aligned}
&\texttt{ATPF} := [\texttt{Tree}_1, \texttt{Tree}_2, \dots, \texttt{Tree}_k];\\
&\texttt{Tree} := \big((\texttt{weight}(\texttt{Tree}), \texttt{weight}(\texttt{Tree}) - \texttt{weight}(\texttt{Tree}_k)), [\texttt{Tree}_1, \texttt{Tree}_2, \dots, \texttt{Tree}_k]\big);\\
&\texttt{Tree} := \big((\texttt{weight}(\texttt{Tree}), \texttt{weight}(\texttt{Tree}) - \texttt{weight}(\texttt{Tree}_k)), [\texttt{Leaf}_1, \texttt{Leaf}_2, \dots, \texttt{Leaf}_k]\big);\\
&\texttt{Leaf} := ((\texttt{weight}(\texttt{Leaf}), 0), l), \text{ where } l \text{ is a list;}
\end{aligned}$$

Note that this function uses the depth of the atpf in order to differentiate between the leaves and the intermediate levels of the tree, which require two different types of treatement.

```
1 def convert_atpf_to_atf(atpf,depth):
2   """ the source code of this function can be found in cata.py """
3   return the_atf
```

The reason for this is that the procedure print_atf (see section 8.4) is to display ascii trees whose trunks are on the left of the screen, as show below.

```
|                       |
|_______________        |
|               |       |
|_______        |       |
|   |   |       |       |
A   C   D...E   F       B
```

Substracting the rightmost weights of the atpf from the weight placed below it, in the tree, allows print_atf to know when it needs to stop printing the horizontal level of the tree. Intuitively, the following pictures shows what atpf would look like without convertion into an atf, where the red underscore symbols sticking out toward the right symbolize the amount of weight subtracted in the atf.

```
|                       |
|_______________        |
|               |       |
|_______-----   |       |
|   |   |       |       |
A   C   D...E   F       B
```

8.4. Description of print_atf

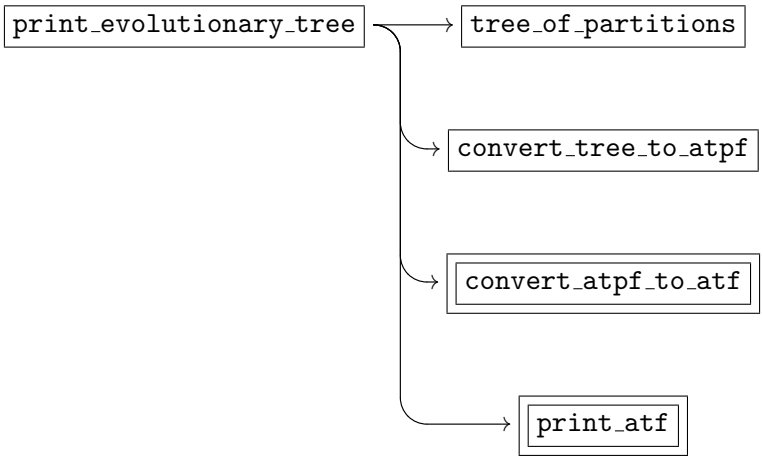
The function print_atf takes an atf and its depth and prints the ascii tree associated with the atf on the standard output.

```
1 def print_atf(atf,depth):
2   """ the source code of this function can be found in patf.py """
```

The following example shows how print_atf can be combined with the procedures convert_tree_to_atpf and convert_atpf_to_atf.

```
>>> l = [[0,1,0,0,0,0], [0,2,0,0,0,1], [0,4,2,3,3,5]]
>>> tree = tree_of_partitions(l)
>>> atpf = convert_tree_to_atpf(tree)
>>> atf = convert_atpf_to_atf(*atpf)
>>> print_atf(atf,atpf[1])
|                       |
|_______________        |
|               |       |
|_______        |       |
|   |   |       |       |
A   C   D...E   F       B
```

8.5. Description of print_evolutionary_tree



The function print_evolutionary_tree takes a list of lists whose pairs of successive lists can be related via MorphismOfPartitions items and returns the tree encoded by this sequence of morphisms.

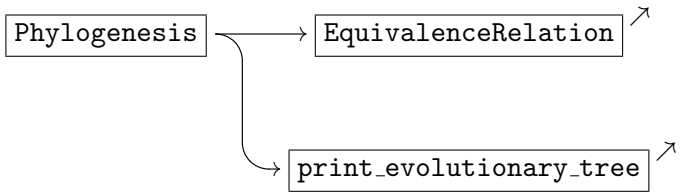
```
1 def print_evolutionary_tree(atf,depth):
2   #Returns a sequence of morphisms of partitions.
3   tree = tree_of_partitions(partitions)
4   #Returns an ascii tree pre-format and its depth.
5   atpf = convert_tree_to_atpf(tree)
6   #Returns the ascii tree format of the atpf.
7   atf = convert_atpf_to_atf(*atpf)
8   #Prints the atf on the standard output.
9   print_atf(atf,atpf[1])
```

See the example given in section 8.5 to see what this procedure does.

Chapter 9

Presentation of the module `Phylogeny.py`

9.1. Description of `Phylogenesis` (class)



The class `Phylogenesis` possesses two objects, namely

- `.taxon` (non-negative integer);
- `.history` (list of lists of indices);

and three methods, namely

- `.__init__` (constructor)
- `.partitions`
- `.print_tree`

A `Phylogenesis` item is meant to be part of another structure called a `Phylogeny` (see section 9.2). The first object `.taxon` of the class `Phylogenesis` stores an integer that allows us to identify the `Phylogenesis` item with respect to other `Phylogenesis` items in the `Phylogeny` structure. On the other hand, the object `.history` stores a list of lists encoding a historical record of the coalescence events between the `Phylogenesis` item and the other taxa contained by the `Phylogeny` structure. The list of lists contained in the object `.history` should:

- start with a singleton list containing the integer representing the taxon itself;
- be such that every list should contain its predecessor list;

```
class Phylogenesis:
  #The objects of the class are:
  #.taxon (non-negative integer);
  #.history (list of lists of indices);
  def __init__(self,history):
  """ the source code of this constructor can be found in cl_pgs.py """
  def partitions(self):
  """ the source code of this function can be found in cl_pgs.py """
    return partitions
  def print_tree(self):
  #Returns the evolutionary tree described by the list of lists outputted
  #by the procedure partitions().
    return print_evolutionary_tree(self.partitions())
```

The constructor `.__init__` takes a non-empty list of lists whose first list is a singleton and allocates

- the index contained in the first internal list of the input list to the object `.taxon`,
- the input list to the object `.history`.

Before terminating, the procedure checks whether the list of lists is made of non-negative integers and whether each list preceding another is contained in the successor list. Below, we give the example of an initialization of a `Phylogenesis` item.

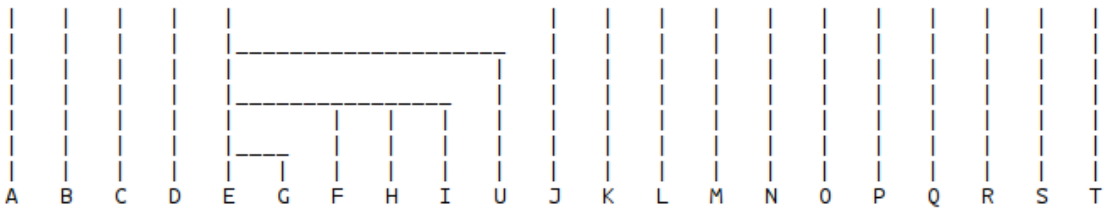
```
>>> history = [[4],[4,6],[5,4,7,8,6],[6,5,4,7,8,20]]
>>> phylogenesis = Phylogenesis(history)
>>> print(phylogenesis.taxon)
4
>>> print(phylogenesis.history[len(phylogenesis.history)-1])
[6, 5, 4, 7, 8, 20]
```

The method `.partitions()` returns the sequence of partitions induced by the list of lists contained in the object `.history` over the set of indices ranging from 0 to the maximum index of the last list of the object `.history`.

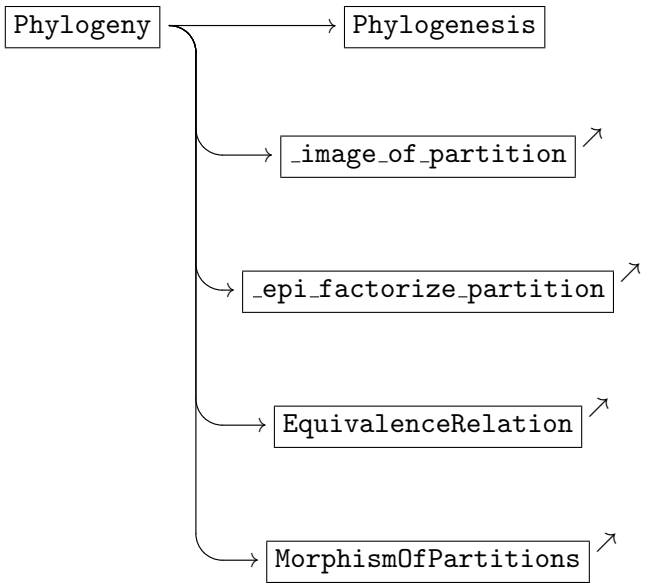
```
>>> p = phylogenesis.partitions()
>>> for i in range(len(p)):
>>>   print(p[i])
[1, 2, 3, 4, 0, 0, 0, 0, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
[1, 2, 3, 4, 0, 0, 0, 0, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
[1, 2, 3, 4, 0, 5, 0, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[1, 2, 3, 4, 0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

The method `.print_tree()` returns the evolutionary tree associated with the sequence of partitions returned by `.partition()` (for more intuition, see section 8).

```
>>> phylogenesis.print_tree()
```



9.2. Description of Phylogeny (class)



The class Phylogeny possesses one object, namely

- .phylogeneses (list of Phylogenesis items)

and ten methods, namely

- .__init__ (constructor)
- .coalescent
- .extend
- .count_uniformity
- .boolean_partition
- .make_friends
- .set_up_friendship
- .score
- .choose
- .set_up_competition

The object .phylogenesis is supposed to contained a list of Phylogenesis items. The taxon associated with the i-th phylogenesis should be indexed by the interger i itself and any label appearing in the Phylogenesis items of the list should have its own Phylogenesis item included in the list.

```
1 class Phylogeny:
2   #The objects of the class are:
3   #.phylogeneses (lists of Phylogenesis items);
4   def __init__(self,phylogeneses):
5   """ the source code of this constructor can be found in cl_pgs.py """
```

9.2.1. First generation. Below, we will often use the term *first generation* of the phylogenesis of a certain taxon t to refer to the last list contained in the list

```
self.phylogeneses[t].history
```

relative to the indexing order associated with the list structure.

9.2.2. Constructor. The constructor .__init__ takes a list of lists of lists containing non-negative integers and use every internal list of the input to create a Phylogenesis item, which is stored in the object .phylogeneses.

The following lines show how a phylogeny can be created via the consutructor of the class.

```
>>> p = list()
>>> for i in range(8):
>>>   p.append([[i], [i, (i+10) % 8], [i, (i+10) % 8, (i+5*i+13) % 8]]))
>>> pgy = Phylogeny(p)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print("taxon:  "+str(pgy.phylogeneses[i].taxon))
>>>   print(pgy.phylogeneses[i].history)
taxon:  0
[[0], [0, 2], [0, 2, 5]]
taxon:  1
[[1], [1, 3], [1, 3, 3]]
taxon:  2
[[2], [2, 4], [2, 4, 1]]
taxon:  3
[[3], [3, 5], [3, 5, 7]]
taxon:  4
[[4], [4, 6], [4, 6, 5]]
taxon:  5
[[5], [5, 7], [5, 7, 3]]
taxon:  6
[[6], [6, 0], [6, 0, 1]]
taxon:  7
[[7], [7, 1], [7, 1, 7]]
```

9.2.3. Elementary methods. Phylogeny items can be updated, modified or analyzed through the methods .coalescent; .extend; and .make_friends, which we present in the following sections.

```
 6   def coalescent(self):
 7   """ the source code of this constructor can be found in cl_pgs.py """
 8     return coalescent
 9   def extend(self,extension):
10   """ the source code of this constructor can be found in cl_pgs.py """
11       return False
12   def make_friends(self,taxon):
13   """ the source code of this constructor can be found in cl_pgs.py """
14     return (friends,coalescence_hypothesis)
```

9.2.4. Coalescent. The method .coalescent() returns the list of the last lists of the objects .history of each Phylogenesis item contained in the object .phylogeneses (*i.e.* what one would like to understand as the *first generations* of the current phylogeny). The k-th list of the outputted list is therefore the first generation (or last list) of the history of taxon k (see example below).

```
>>> coalescent = pgy.coalescent()
>>> for i in range(len(coalescent)):
>>>   print("1st generation of " + str(i) + "'s history:  " + " " +
    str(coalescent[i]))
1st generation of 0's history:  [0, 2, 5]
1st generation of 1's history:  [1, 3, 3]
1st generation of 2's history:  [2, 4, 1]
1st generation of 3's history:  [3, 5, 7]
1st generation of 4's history:  [4, 6, 5]
1st generation of 5's history:  [5, 7, 3]
1st generation of 6's history:  [6, 0, 1]
1st generation of 7's history:  [7, 1, 7]
```

9.2.5. Extend. The method `.extend` takes a list of pairs of the form `(t,l)` where `t` is the label of a taxon of the `Phylogeny` item (accessible through `pgy.phylogeneses[−].taxon`) and `l` is a list of taxa and it updates the object `.phylogeneses` as follows:

- for all pairs `(t,l)` contained in the input passed to `.extend`:
  1) ▷ if *every* list `l` contains the last list of `self.phylogeneses[t].history`,
     ▷ if *at least one* of the lists `l` strictly contains the last list of
     `self.phylogeneses[t].history`,
     → then every list `l` is appended to the list `self.phylogeneses[t].history` and the value `True` is returned;
  2) if there is no strict inclusion of the last list of `self.phylogeneses[t].history` into `l`, then the object `.phylogeneses` is not modified and the value `False` is returned;
  3) otherwise, an error message is returned and the procedure exit the program;
- if an update has happened, then for all other taxa `t` of the phylogeny that do not appear in the input of `.extend`, the last list of `self.phylogeneses[t].history` (*i.e.* the first generation of the history of the phylogenesis of `t`) is again repeated (*i.e.* appended again) in the list `self.phylogeneses[t].history`.

The following code lines illustrate these various cases for the example used in the previous sections (starting from section 9.2.2).

```
>>> extension = [(5,[3,7,7,5]),(7,[7,1])]
>>> flag = pgy.extend(extension)
>>> print(flag)
False
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3]]
[[2], [2, 4], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3]]
[[6], [6, 0], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7]]
>>> extension = [(5,[3,7,7,5]),(7,[7,1]),(1,[1,3,4,5])]
>>> flag = pgy.extend(extension)
>>> print(flag)
```

```
True
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
>>> extension = [(-1,["error"])]
>>> pgy.extend(extension)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
>>> extension = [(5,[1,4,80]),(7,["error"])]
>>> pgy.extend(extension)
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
Error:  in Phylogeny.extend:  the extension is not compatible with the
phylogenesis of taxon 5
```

9.2.6. Make friends. The method `.make_friends` takes the label of a taxon (i.e. a non-negative integer) and returns a pair of lists `(friends,hypothesis)` where

- the list `friends` contains all those taxa that have not coalesced with the input taxon, which means that there are not in the first generation of the phylogenesis of the taxon;
- the list `hypothesis` contains the (sorted) lists obtained by making the union of the first generation of the input taxon with the first generation of one of the taxon in `friends`.

For illustration, consider the phylogeny constructed in section 9.2.5. Below, we first recall the structure of that phylogeny.

```
>>> for i in range(len(pgy.phylogeneses)):
>>>   print(pgy.phylogeneses[i].history)
[[0], [0, 2], [0, 2, 5], [0, 2, 5]]
[[1], [1, 3], [1, 3, 3], [1, 3, 4, 5]]
[[2], [2, 4], [2, 4, 1], [2, 4, 1]]
[[3], [3, 5], [3, 5, 7], [3, 5, 7]]
[[4], [4, 6], [4, 6, 5], [4, 6, 5]]
```

```
[[5], [5, 7], [5, 7, 3], [3, 7, 5]]
[[6], [6, 0], [6, 0, 1], [6, 0, 1]]
[[7], [7, 1], [7, 1, 7], [7, 1]]
```

The output of the method .make_friends for taxa 1, 2 and 7 is as follows.

```
>>> friends_made = pgy.make_friends(1)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 1 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 1 and 0 have ancestor [0, 1, 2, 3, 4, 5]
taxa 1 and 2 have ancestor [1, 2, 3, 4, 5]
taxa 1 and 6 have ancestor [0, 1, 3, 4, 5, 6]
taxa 1 and 7 have ancestor [1, 3, 4, 5, 7]
>>> friends_made = pgy.make_friends(2)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 2 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 2 and 0 have ancestor [0, 1, 2, 4, 5]
taxa 2 and 3 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 5 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 6 have ancestor [0, 1, 2, 4, 6]
taxa 2 and 7 have ancestor [1, 2, 4, 7]
>>> friends_made = pgy.make_friends(7)
>>> friends = friends_made[0]
>>> coalescence_hypothesis = friends_made[1]
>>> for i in range(len(friends)):
>>>   print("taxa 7 and "+ str(friends[i])+" have ancestor
    "+str(coalescence_hypothesis[i]))
taxa 7 and 0 have ancestor [0, 1, 2, 5, 7]
taxa 7 and 2 have ancestor [1, 2, 4, 7]
taxa 7 and 3 have ancestor [1, 3, 5, 7]
taxa 7 and 4 have ancestor [1, 4, 5, 6, 7]
taxa 7 and 5 have ancestor [1, 3, 5, 7]
taxa 7 and 6 have ancestor [0, 1, 6, 7]
```

9.2.7. Algorithm for constructing phylogenies. The next set of methods are procedures meant to be used to implement the algorithm described in [4], which constructs a phylogeny according ot the definition given thereof.

```
15   def set_up_friendships(self):
16   """ the source code of this constructor can be found in cl_pgs.py """
17     return (friendships,coalescence_hypotheses)
15   def score(self,partitions,friendship_network):
16   """ the source code of this constructor can be found in cl_pgs.py """
17     return score_cardinality_adjusted
18   def choose(self,scores):
19   """ the source code of this constructor can be found in cl_pgs.py """
20     return result
```

```
21    def set_up_competition(self,best_fit):
22    """ the source code of this constructor can be found in cl_pgs.py """
23      return coalescence_hypothesis
```

9.2.8. Set up friendships. The method `.set_up_friendships()` returns a pair of lists `(friendships,hypotheses)` containing the lists of the two different outputs of the method `.make_friends` for every taxon of the phylogeny. More specifically,

- `friendships` is the list of lists whose t-th list contains the first output of the procedure `self.make_friends` for taxon t;
- `hypotheses` is the list of lists whose t-th list contains the second output of the procedure `self.make_friends` for taxon t;

The following code lines give a description of the output of `.set_up_friendships()` for the phylogeny used in section 9.2.6.

```
>>> friendships_made = pgy.set_up_friendships()
>>> friendships = friendships_made[0]
>>> coalescence_hypotheses = friendships_made[1]
>>> for t in range(len(friendships)):
>>>   for r in range(len(friendships[t])):
>>>     print("taxa " + str(t) + " and " + str(friendships[t][r])+" have
    ancestor "+str(coalescence_hypotheses[t][r]))
taxa 0 and 1 have ancestor [0, 1, 2, 3, 4, 5]
taxa 0 and 3 have ancestor [0, 2, 3, 5, 7]
taxa 0 and 4 have ancestor [0, 2, 4, 5, 6]
taxa 0 and 6 have ancestor [0, 1, 2, 5, 6]
taxa 0 and 7 have ancestor [0, 1, 2, 5, 7]
taxa 1 and 0 have ancestor [0, 1, 2, 3, 4, 5]
taxa 1 and 2 have ancestor [1, 2, 3, 4, 5]
taxa 1 and 6 have ancestor [0, 1, 3, 4, 5, 6]
taxa 1 and 7 have ancestor [1, 3, 4, 5, 7]
taxa 2 and 0 have ancestor [0, 1, 2, 4, 5]
taxa 2 and 3 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 5 have ancestor [1, 2, 3, 4, 5, 7]
taxa 2 and 6 have ancestor [0, 1, 2, 4, 6]
taxa 2 and 7 have ancestor [1, 2, 4, 7]
taxa 3 and 0 have ancestor [0, 2, 3, 5, 7]
taxa 3 and 1 have ancestor [1, 3, 4, 5, 7]
taxa 3 and 2 have ancestor [1, 2, 3, 4, 5, 7]
taxa 3 and 4 have ancestor [3, 4, 5, 6, 7]
taxa 3 and 6 have ancestor [0, 1, 3, 5, 6, 7]
taxa 4 and 0 have ancestor [0, 2, 4, 5, 6]
taxa 4 and 1 have ancestor [1, 3, 4, 5, 6]
taxa 4 and 2 have ancestor [1, 2, 4, 5, 6]
taxa 4 and 3 have ancestor [3, 4, 5, 6, 7]
taxa 4 and 7 have ancestor [1, 4, 5, 6, 7]
taxa 5 and 0 have ancestor [0, 2, 3, 5, 7]
taxa 5 and 1 have ancestor [1, 3, 4, 5, 7]
taxa 5 and 2 have ancestor [1, 2, 3, 4, 5, 7]
taxa 5 and 4 have ancestor [3, 4, 5, 6, 7]
taxa 5 and 6 have ancestor [0, 1, 3, 5, 6, 7]
```

```
taxa 6 and 2 have ancestor [0, 1, 2, 4, 6]
taxa 6 and 3 have ancestor [0, 1, 3, 5, 6, 7]
taxa 6 and 4 have ancestor [0, 1, 4, 5, 6]
taxa 6 and 5 have ancestor [0, 1, 3, 5, 6, 7]
taxa 6 and 7 have ancestor [0, 1, 6, 7]
taxa 7 and 0 have ancestor [0, 1, 2, 5, 7]
taxa 7 and 2 have ancestor [1, 2, 4, 7]
taxa 7 and 3 have ancestor [1, 3, 5, 7]
taxa 7 and 4 have ancestor [1, 4, 5, 6, 7]
taxa 7 and 5 have ancestor [1, 3, 5, 7]
taxa 7 and 6 have ancestor [0, 1, 6, 7]
```

9.2.9. Scoring system. The method `.score` takes a list of lists of non-negative integers (i.e. partitions), call it `partitions`, and a pair of lists, say `(friendships,hypotheses)`, where

- `friendships` is a list of lists;
- `hypotheses` is a list of length `len(friendships)` whose `t`-th element is a list of length `len(friendships[t])` whose elements are lists of integers ranging from `0` to

  `len(self.phylogeneses)-1`

  (preferrably sorted from smallest to greatest);

and returns a list of length `len(friendships)` whose `t`-th element is a list of triples of the form `(r,large,exact)` where

- `r` runs over the elements of `friendships[t]` ,
- `large` is the large score [4] of the hypothetical ancestor `hypotheses[t][r]` within the set of ancestors contained in `hypotheses[t]` for the list of partitions given in the input,
- `exact` is the exact score [4] of the hypothetical ancestor `hypotheses[t][r]` within the set of ancestors contained in `hypotheses[t]` for the list of partitions given in the input.

This means that `large` is the number of partitions belonging to the first input list `partitions` for which there is a morphism of partitions `x.quotient()` $\to$ `partitions[i]` where we take

`x = EquivalenceRelation([hypotheses[t][r]],len(self.phylogeneses)-1)`

and `exact` is the number of partitions that were counted in the large score of `r` such that if these partitions belong to the large score of any other element `s` in `friendships[t]`, then either the equality

`hypotheses[t][r] = hypotheses[t][s]`

holds or the intersection of `hypotheses[t][r]` with `hypotheses[t][s]` is empty.

The second input of the method `.score` can, for instance, be taken to be the output of the procedure

`self.set_up_friendships()`.

For example, consider the following file containing a sequence alignment.

Align.fa

```
>Alice
ACGCTAGCGCGATCGATCGGATCGATCGATC
>Bob
ACGACTTAGCGGATCTGATACTCCCTCGATC
>Carles
ACGACCTAGCGGATCTTATAACTCACCGATC
>Doug
ACGCTAGCGGCTGATAACGATCGTATCGATC
>Eric
ACGCATGCGCGATCGACGGATCGTATCTATC
>Fred
ACGACCTAGCAGATTTCTAATCTCAACGATC
>Gary
ACGCGAGCATCTGAACACGATTGTAACGATC
>Haley
ACGCTACGCGACGATCGGCTTTAGATCGATC
```

Let us now construct the list `local` of partitions induced by each non-trivial column of the previous alignment, as shown below.

```
>>> from efp import _epi_factorize_partition
>>> pre_local = usf.fasta("Align1.fa")[1]
>>> local = list()
>>> for j in range(len(pre_local[0])):
...     column = list()
...       for i in range(len(pre_local)):
...         column.append(pre_local[i][j])
...       r = _epi_factorize_partition(column)
...       if r != [0]*len(column):
...         local.append(r)
...
```

Then, we can use the method `.score`, as shown below, with the `Phylogeny` item `pgy` considered in section 9.2.6.

```
>>> l = pgy.score(local,pgy.set_up_friendships())
>>> for t in range(len(l)):
>>>   for (r,large,exact) in l[t]:
>>>     print("taxa " + str(t) + " and " + str(r)+" coalesce with large
    score "+str(large)+" and with exact score "+str(exact))
taxa 0 and 1 coalesce with large score 1 and with exact score 1
taxa 0 and 3 coalesce with large score 2 and with exact score 0
taxa 0 and 4 coalesce with large score 1 and with exact score 0
taxa 0 and 6 coalesce with large score 1 and with exact score 0
taxa 0 and 7 coalesce with large score 1 and with exact score 0
taxa 1 and 0 coalesce with large score 1 and with exact score 0
taxa 1 and 2 coalesce with large score 2 and with exact score 1
taxa 1 and 6 coalesce with large score 0 and with exact score 0
taxa 1 and 7 coalesce with large score 0 and with exact score 0
```

```
taxa 2 and 0 coalesce with large score 2 and with exact score 2
taxa 2 and 3 coalesce with large score 0 and with exact score 0
taxa 2 and 5 coalesce with large score 0 and with exact score 0
taxa 2 and 6 coalesce with large score 0 and with exact score 0
taxa 2 and 7 coalesce with large score 0 and with exact score 0
taxa 3 and 0 coalesce with large score 2 and with exact score 0
taxa 3 and 1 coalesce with large score 0 and with exact score 0
taxa 3 and 2 coalesce with large score 0 and with exact score 0
taxa 3 and 4 coalesce with large score 2 and with exact score 1
taxa 3 and 6 coalesce with large score 1 and with exact score 0
taxa 4 and 0 coalesce with large score 1 and with exact score 0
taxa 4 and 1 coalesce with large score 1 and with exact score 0
taxa 4 and 2 coalesce with large score 1 and with exact score 0
taxa 4 and 3 coalesce with large score 2 and with exact score 1
taxa 4 and 7 coalesce with large score 0 and with exact score 0
taxa 5 and 0 coalesce with large score 2 and with exact score 0
taxa 5 and 1 coalesce with large score 0 and with exact score 0
taxa 5 and 2 coalesce with large score 0 and with exact score 0
taxa 5 and 4 coalesce with large score 2 and with exact score 1
taxa 5 and 6 coalesce with large score 1 and with exact score 0
taxa 6 and 2 coalesce with large score 0 and with exact score 0
taxa 6 and 3 coalesce with large score 1 and with exact score 0
taxa 6 and 4 coalesce with large score 0 and with exact score 0
taxa 6 and 5 coalesce with large score 1 and with exact score 0
taxa 6 and 7 coalesce with large score 2 and with exact score 1
taxa 7 and 0 coalesce with large score 1 and with exact score 0
taxa 7 and 2 coalesce with large score 0 and with exact score 0
taxa 7 and 3 coalesce with large score 1 and with exact score 0
taxa 7 and 4 coalesce with large score 0 and with exact score 0
taxa 7 and 5 coalesce with large score 1 and with exact score 0
taxa 7 and 6 coalesce with large score 2 and with exact score 1
```

9.2.10. Choose. The method `.choose` takes a list of lists of triples `(r,l,e)` where `l` and `e` are non-negative integers and returns a list of lists whose i-th list is the list of those elements `r` of the i-th internal list of the input list for which the associated pairs `(l,e)` are equal to the greatest local maxima of the function Γ : `(e,l)` $\mapsto$ `(l,e)` ordered by the lexicographical order and relative to the pairs of the i-th internal list of the input list (see the example below and [4, Definition 3.14]).

```
>>> k1 = [[("a",1,0),("b",2,0),("c",1,1),("d",3,3)]]
>>> choose = pgy.choose(k1)
>>> print(choose)
[['d']]
>>> k2 = [[("a",1,0),("b",2,0),("c",1,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k2)
>>> print(choose)
[['d', 'e']]
>>> k3 = [[("a",1,0),("b",2,0),("c",8,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k3)
>>> print(choose)
[['c']]
```

```
>>> k4 = [[("a",1,0),("b",9,0),("c",8,1),("d",3,3),("e",3,3)]]
>>> choose = pgy.choose(k4)
>>> print(choose)
[['b']]
>>> k = k1 + k2 + k3 + k4
>>> choose = pgy.choose(k)
>>> print(choose)
[['d'], ['d', 'e'], ['c'], ['b']]
```

The following example shows the type of output that one gets if one gives the list `l` of the example of section 9.2.9 to the method `.choose`.

```
>>> choose = pgy.choose(l)
>>> for t in range(len(choose)):
>>>   print("Closest relatives of taxon " + str(t) + " are " +
    str(choose[t]))
Closest relatives of taxon 0 are [3]
Closest relatives of taxon 1 are [2]
Closest relatives of taxon 2 are [0]
Closest relatives of taxon 3 are [4]
Closest relatives of taxon 4 are [3]
Closest relatives of taxon 5 are [4]
Closest relatives of taxon 6 are [7]
Closest relatives of taxon 7 are [6]
```

9.2.11. Set up competition. The method `.set_up_competition` takes a list of lists of integers whose length must be equal to the length of `self.phylogeneses` (i.e. the number of taxa of the phylogeny) and returns a list of lists of integers whose length is also equal to the length of `self.phylogeneses` and whose t-th internal list is the union of the t-th list of `self.coalescent()` with the `r`-th list of `self.coalescent()` for every element `r` in the t-th internal list of the input list.

The following example shows the ouput of the method `.set_up_competition` when it is given the list `choose` constructed in the last example of section 9.2.10.

```
>>> competition = pgy.set_up_competition(choose)
>>> for t in range(len(competition)):
>>>   print("Next competing generation representing taxon " + str(t) + " is
    " + str(competition[t]))
Next competing generation representing taxon 0 is [0, 2, 3, 5, 7]
Next competing generation representing taxon 1 is [1, 2, 3, 4, 5]
Next competing generation representing taxon 2 is [0, 1, 2, 4, 5]
Next competing generation representing taxon 3 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 4 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 5 is [3, 4, 5, 6, 7]
Next competing generation representing taxon 6 is [0, 1, 6, 7]
Next competing generation representing taxon 7 is [0, 1, 6, 7]
```

Bibliography

[1] R. Tuyéras, (2017), *Category theory for genetics*, arXiv:1708.05255

[2] R. Tuyéras, (2018), *Category theory for genetics I: mutations and sequence alignments*, arXiv:1805.07002

[3] R. Tuyéras, (2018), *Category theory for genetics II: genotype, phenotype and haplotype*, arXiv:1805.07004

[4] R. Tuyéras, (2018), *Category theory for genetics III: natural selection, evolution and phylogeny.*