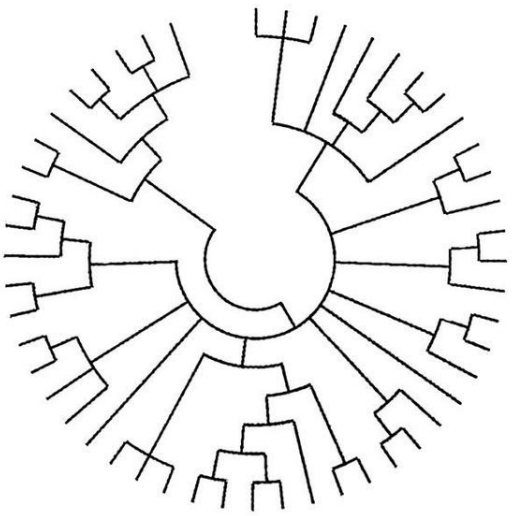
DOCUMENTATION FOR THE PYTHON LIBRARY PEDIGRAD.PY

Rémy Tuyéras
Department of Mathematics
Massachusetts Institute of Technology

Version 1.1



Typeset in LaTeX 2$_\varepsilon$.

Contents

Introduction

1.1. About pedigrads and `Pedigrad.py`

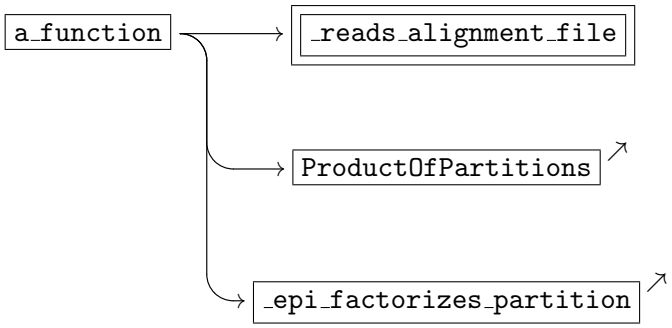
Pedigrads are mathematical tools that were initially created to model various mechanisms of genetics (see [2]). Mathematically, pedigrads are defined as functors sending the cones of a certain type of limit sketch to cones in a given category of values. Pedigrads can encode different aspects of biology depending on their 'categories of values' and associated cones. So far, the python library `Pedigrad.py` only includes pedigrads taking their values in the category of partitions whose cones are product cones (see [1]).

1.2. About this documentation

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

- `PartitionCategory.py` (see Chapter 3)
- `SegmentCategory.py` (see Chapter 4)
- `PedigradCategory.py` (see Chapter 5)
- `AsciiTree.py` (see Chapter 6)

A description of a function will always start with a dependcy flow chart as given below if the function depends on other procedures.



A double box indicates that the function does not have any dependencies while an arrow on the top-right corner of a box means that the function belongs to another module of the library.

Also, the python code will always be specified in text editor mode as follows.

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

A console mode is also used for most of the examples (see below).

```
>>> print(P.loci)
[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. Background

[Recall content from previous papers]

2.2. Pre-ordered sets

2.3. Categories of segments

2.4. Categories of Partitions

2.5. Categories of Pedigrads

Presentation of the module `PartitionCategory.py`

3.1. Description of `_image_of_partition`

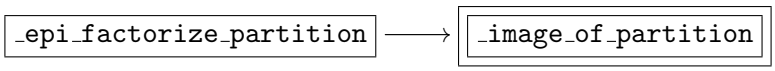
The function `_image_of_partition` takes a list of elements and returns the list of its elements without repetition in the order in which they can be accessed from the left to the right.

```
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']
```

3.2. Description of `_epi_factorize_partition`



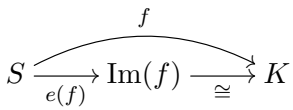
The function `_epi_factorize_partition` relabels the elements of a list with non-negative integers. It starts with the integer 0 and attributes a new label by increasing the previously attributed label by 1. The first element of the list always receives the label 0 and the highest integer used in the relabeling equals the length of the image (section 3.1) 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 returs a canonical *choice* of epimorphism.

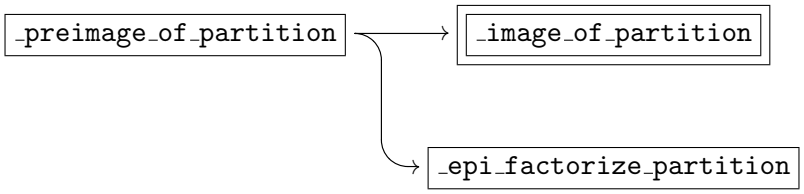


This choice ensures that two partitions characterized by the same set of universal properties are *equal* as python lists.

```
>>> 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.

3.3. 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, the returned list the_preimage (see 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.

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

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 $e(f) : S \to \mathrm{Im}(f)$ whose codomain is equipped with the natural order on integers (see section 3.2).

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

```
>>> 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))
```

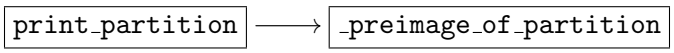
```
[[0, 6], [1, 7], [2, 8], [3], [4], [5]]
>>> print(_epi_factorize_partition(p))
[0, 1, 2, 3, 4, 5, 0, 1, 2]
```

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 orders its fibers with respect to the order in which the elements of the input list appear.

3.4. Description of print_partition

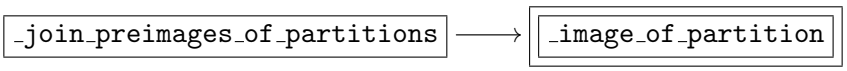


The function `print_partition` is a debug function that takes a list of elements and prints its preimage on the standard output (i.e. the console).

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

See section 3.3 for examples.

3.5. 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) 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):
2   """ the source code of this function can be found in jpop.py """
3   return the_join
```

For instance, the two input lists `preimage1` and `preimage2` could be two outputs of the procedure

$$\texttt{\_preimage\_of\_partition}(-,-)$$

for two input lists of the same length. Form the point of view of partitions, this would amount to computing the coproduct of two partitions and taking its pre-image. Since a category of partitions can be seen as a partially ordered set, such a coproduct is also the *join* of the two partitions.

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

In terms of implementation, the program

(3.1) `_join_preimages_of_partitions(p1,p2)`

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 (3.1) 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]]
```

The element 0 has now been found in p2 and does not need to be searched again (breaks)

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 (breaks)

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)
>>> print(closure_of_a)
[[15, 18, 0, 13, 1, 2, 3, 5, 4, 6], [23, 12, 7, 11, 22]]
```

3.6. 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)
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 is second input 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, for its part, stored in the object `.range`.

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

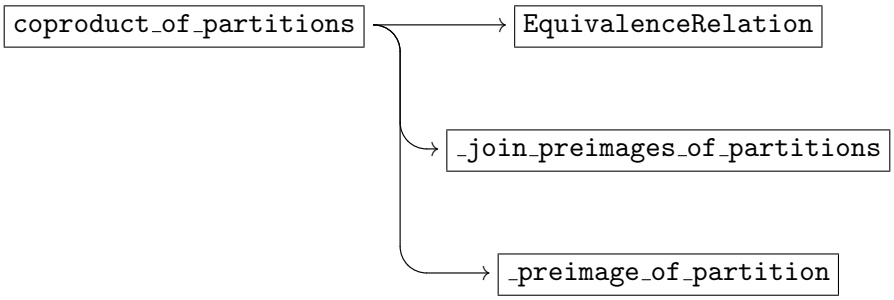
The method `.closure()` replaces the content of the object `.classes` with the transitive closure of its classes. After this procedure, the object `.classes` describes an actual equivalence relation (modulo the singleton equivalence classes, which do not need to be specified for obvious reasons).

```
>>> 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]
```

3.7. Description of coproduct_of_partitions



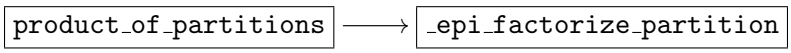
The function coproduct_of_partitions takes two lists of the same legnth and returns their coproduct (or join) as partitions. Specifically, the procedure outputs the quotient of the join of their preimages. If the two input lists do not have the same length, then an error message is outputted and the 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)))
 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 3.5).

```
>>> 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]
```

3.8. Description of product_of_partitions



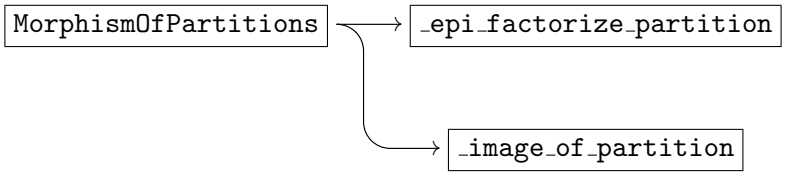
The function product_of_partitions takes two lists and returns a list that is the relabeling of the zipping of the two lists (i.e. the list of pairs of elements with corresponding indices in each of the input lists) via the procedure _epi_factorize_partition.

```
 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.
```

3.9. 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 and stores, in the object .arrow, a list that describes, if it exists, the (unique) morphism of partitions from the first list (seen as a partition) to the second list (seen as a partition). The canonical epimorphisms associated with the partitions of the first and second input lists (see section 3.2) 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):
7   """ the source code of this constructor can be found in cl_mop.py """
```

If we suppose that the two input lists are labeled in the same way as the procedure

_epi_factorize_partition

would (re)label them, then the list that is to be contained in the object .arrow is the zipping of the two lists, as illustrated in the following example.

Consider the following lists.

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

Their zipping is as follows:

```
>>> 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 outputs the list recapturing the previous graph.

```
>>> 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)]
```

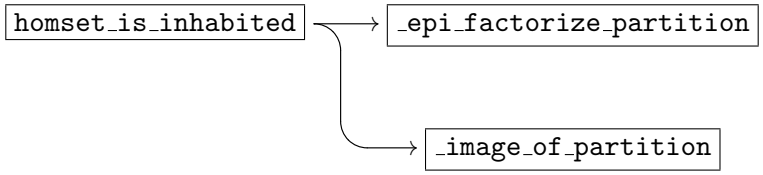
As can be seen, the argument 3 is 'mapped' to two different images, namely 3 and 6. In this case, the constructor .__init__ exits the program with an error message.

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.
```

Note that the function of section 3.10 can be used to avoid the earlier error message.

3.10. Description of homset_is_inhabited



The function **homset_is_inhabited** takes two lists and returns a Boolean value indicating whether there exists a morphism of partitions from the underlying partition of the first input list to the underlying partition of the second input list.

```
1 def homset_is_inhabited(source,target):
2   """ the source code of this function can be found in hii.py """
3   return flag
```

The way this function is implemented is very similar to the way the contructor of MorphismOfPartitions is implemented (see section 3.9).

```
>>> p1 = [0,1,2,3,3,4,5]
>>> p2 = [0,1,2,3,3,3,1]
>>> homset_is_inhabited(p1,p2)
True
>>> p2 = [0,1,2,3,6,3,1]
>>> homset_is_inhabited(p1,p2)
False
```

Chapter 4

Presentation of the module SegmentCategory.py

4.1. Description of _read_pre_order

The function **_read_pre_order** takes the name of a file (called **name_of_file**) and returns a Boolean value and a list of lists.

```
1 def _read_pre_order(name_of_file):
2   """ the source code of this function can be found in rpo.py """
3   return (flag_zero_obj,the_pre_order)
```

The input file is supposed to contain some specific code lines that describe a pre-ordered set. The code follows a certain syntax that will be detailed in this section. Because this syntax is compatible with the yaml grammar, these pre-ordered sets will usually be specified with the extension **.yml** in our examples. Here is an example of such a file.

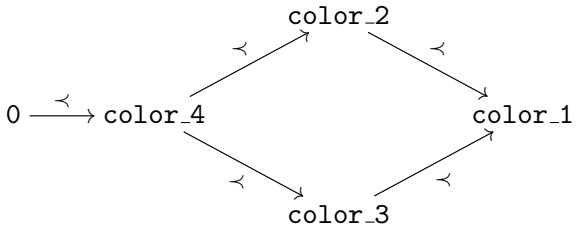
Omega.yml

```
 1 #This is an example of pre-ordered set.
 2
 3 !obj:
 4 - color_1 #object for color 1
 5 - color_2 #object for color 2
 6 - color_3 #object for color 3
 7 - color_4 #object for color 4
 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_4;
13 - color_2 > color_4;
```

Diagrammatically, this pre-ordered set can be described by the graph given below, whose square commutes. We will see below why there is an additional object **0** that is not specified

in the yaml file.

(4.1)



4.1.1. Comments. Comments are allowed and are specified as in python and in yaml, which is to say by the character `#`.

4.1.2. Objects. The set of elements of the pre-ordered set should always be specified in terms of a list of words succeeding one of the key words '`obj:`' or '`!obj:`' as shown below.

```
obj:[name1][separator][name2][separator] (etc.)
!obj:[name1][separator][name2][separator] (etc.)
```

The labels `[name1]` and `[name2]` stand for words that should only be made of the characters `0-9`, `@`, `A-Z`, `_` and `a-z` while the label `[separator]` stands for any character that is not in this list of characters. Repetitions of names are taken care by the parser, which ignores all repetitions. The following specification of objects is equivalent to that given in `Omega.yml`.

Objects.yml

```
1 #Note that we now use different types of separator.
2 !obj:  color_1 ;
3 ,, color_2 ; color_1 ;
4 --> color_3.!color_4 *
```

While the key word `obj:` is only supposed to present the list of elements of the pre-ordered set, the key word `!obj:` should only be used to formally *add* a formal initial object (i.e. a mimimum element) to the pre-ordered set. This element was represented by `0` in diagram (4.1).

4.1.3. Relations. The pre-order relations of the set should be specified after the key word `rel:` by a list of phrases satisfying the following syntax.

```
[dominant_object]>[predecessor1][separator][predecessor2][separator] (etc.);
```

Such a line means that `[dominant_object]` is greater than or equal to all the elements listed after the symbol '`>`'. The list of predecessors stops at the symbol '`;`'. Note that the symbol '`;`' can be omitted for the last line of the file. The following specification of relations is equivalent to that given in `Omega.yml`.

Omega_bis.yml

```
1 #The pre-ordered set can be very compact.
2 !obj:  1 2 3 4
3 rel:  1>2,3 ; 3>4 ; 2>4
```

The labels `[dominant_object]` and `[predecessor(`n`)]` stand for words that should only be made of the characters `0-9`, `@`, `A-Z`, `_` and `a-z` while the label `[separator]` stands for any character that is not in this list of characters.

4.1.4. Output. Finally, the output of the function `_read_pre_order` is a pair

```
(flag_zero_obj,the_pre_order)
```

whose first component `flag_zero_obj` is a Boolean value specifying by `False` or `True` whether the word `obj:` (`False`) or `!obj:` (`True`) was used to define the list of objects and whose

second component the_pre_order is a list of lists giving the *non-transitive* down-closures of the elements of the pre-ordered set in the order in which these were given in the input file.

Specifically, a list in the list the_pre_order will always start with the element of which it is supposed to give the down-closure, as shown below.

```
[dominant_object, predecessor1, predecessor2, (etc.)]
```

For instance, the ouput of our previous example Omega.yml is as follows.

```
>>> omega = _read_pre_order("Omega.yml")
>>> print("Formal initial object:  "+str(omega[0]))
>>> for i in range(len(omega[1])):
...   print(omega[1][i])
Formal initial object:  True
['color_1', 'color_2', 'color_3']
['color_2', 'color_4']
['color_3', 'color_4']
['color_4']
```

4.2. Description of _transitive_closure

The function _transitive_closure takes a list of lists (called down_closure) and returns another list of lists whose pointer is the same as that given in the input (i.e. the input is therefore modified by the function to make the output).

```
1 def _transitive_closure(down_closure):
2   """ the source code of this function can be found in tc.py """
3   return down_closure
```

The input down_closure is supposed to take the form of a list of the (potentially) non-transitive down-closures of the elements of a given pre-ordered set. In other words, the input down_closure should take the same form as that taken by the second outputs of the procedure _read_pre_order(−) (see section 4.1). Specifically, the down-closure of an element dominant_object is the unique list of down_closure starting with this element (see below).

```
[dominant_object, predecessor1, predecessor2, (etc.)]
```

The output down_closure is then the list of the transitive down-closures of the elements of that pre-ordered set.

Here is an example with the procedure _read_pre_order. Let us consider the following pre-order specification.

Omega.yml

```
1 #This is a non-transitive presentation of a pre-ordered set.
2
3 #The list of elements of the set (without the formal minimum).
4 !obj:  1 2 3 4 5 6
5
6 #The list of non-trivial (generating) relations.
7 rel:  1 > 2; 2 > 3; 3 > 5; 5 > 6;
```

As shown below, we can see that the output of the procedure _read_pre_order does not include the transitive relation 1 > 2 > 3 in the down-closure of 1. On the other hand, the relation 1 > 3 does appear in the down-closure of 1 even it is not specified in Omega.yml.

```
>>> preorder = _read_pre_order("Omega.yml")
>>> print("Formal initial object:  "+str(preorder[0]))
>>> for i in range(len(preorder[1])):
...   print(preorder[1][i])
['1', '2']
['2', '3']
['3', '5']
['4']
['5', '6']
['6']
>>> omega = _transitive_closure(preorder[1])
>>> for i in range(len(omega)):
...   print(omega[i])
['1', '2', '3', '5', '6']
['2', '3', '5', '6']
['3', '5', '6']
['4']
['5', '6']
['6']
```

4.3. Description of SegmentObject (class)

The class SegmentObject possesses two objects, namely

- .topology (list of 2-tuples);
- .colors (list of indices),

and two methods, namely

- .__init__ (constructor)
- .patch

The idea of behind the two objects .colors and .topology is that they contain all the needed information to define a mathetical segment, that is to say a pair (t, c) where t is the *topology*, usually encoded by an order preserving surjection, and c is the *coloring map*, usually encoded by a function going to a pre-ordered set.

```
1 class SegmentObject:
2   #The objects of the class are:
3   #.topology (list of 2-tuples);
4   #.colors (list of indices);
5   def __init__(self,topology,colors):
6   """ the source code of this constructor can be found in cl_so.py """
7   def patch(self,position):
8   """ the source code of this function can be found in cl_so.py """
9     return the_index
```

The constructor .__init__ takes two lists, specifically a lists of indices and a list of pairs of increasing indices, and allocate them in the object .colors and .topology, respectively

For instance, we could first define an uncolored topology of the form

$$(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)$$

by using the list t defined below.

```
>>> t = list()
>>> for i in range(5):
...   t = t + [(3*i,3*i+2)]
```

Then, we could add colors to this topology, say to form the following Boolean segment.

$$(\circ\circ\circ)(\bullet\bullet\bullet)(\bullet\bullet\bullet)(\circ\circ\circ)(\circ\circ\circ)$$

In this case, the colors would be specified by a list of colors whose length is equal to `len(t)` and the associated segment would be defined via the constructor `.__init__` as follows.

```
>>> c = [0,1,1,0,0]
>>> s = SegmentObject(t,c)
```

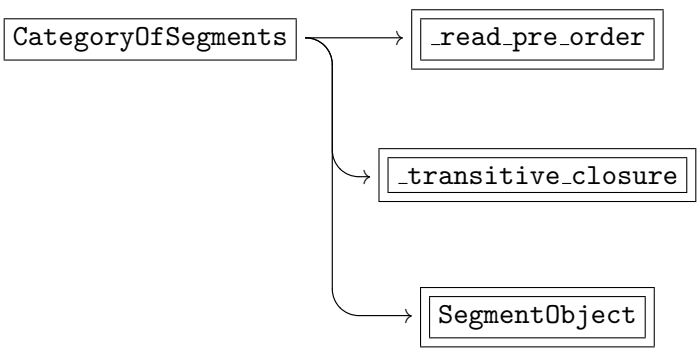
For its part, the method `.patch` takes an integer and returns the index of the first pair `(a,b)` of `.topology` that bounds the integer, that is to say that the input integer is greater than or equal to the first component `a` and also is less than or equal to the second component `b`. If no such index exists, then the procedure returns `-1`.

For example, we can use the method `.patch` to recover all the information contained in our previous segment `s`, as illustrated below.

```
>>> for i in range(15):
...   print("Position "+str(i)+" belongs to "+str(s.topology[s.patch(i)])+"
    and its color is "+str(s.colors[s.patch(i)]))
Position 0 belongs to (0, 2) and its color is 0
Position 1 belongs to (0, 2) and its color is 0
Position 2 belongs to (0, 2) and its color is 0
Position 3 belongs to (3, 5) and its color is 1
Position 4 belongs to (3, 5) and its color is 1
Position 5 belongs to (3, 5) and its color is 1
Position 6 belongs to (6, 8) and its color is 1
Position 7 belongs to (6, 8) and its color is 1
Position 8 belongs to (6, 8) and its color is 1
Position 9 belongs to (9, 11) and its color is 0
Position 10 belongs to (9, 11) and its color is 0
Position 11 belongs to (9, 11) and its color is 0
Position 12 belongs to (12, 14) and its color is 0
Position 13 belongs to (12, 14) and its color is 0
Position 14 belongs to (12, 14) and its color is 0
```

Finally, note that the pre-ordered set that is usually associated with a segment (t, c) can be specified through the class `CategoryOfSegments` defined in section 4.4.

4.4. Description of `CategoryOfSegments` (class)



The class `CategoryOfSegments` possesses three objects, namely

- `.domain` (integer)
- `.mask` (Boolean)
- `.preorder` (list of lists)

and five methods, namely

- `.__init__` (constructor)
- `.can_switch_to`
- `.homset_is_inhabited`
- `.topology`
- `.segment`

In addition, the class `CategoryOfSegments` may be used as a super class (this is allowed by an argument `object`; see below). In this library, the class `CategoryOfSegments` is the paren of the classes `LocalAnalysis` and `Pedigrad` (see section 5.3 and section 5.5).

```
1 class CategoryOfSegments(object):
2   #The objects of the class are:
3   #.domain (integer);
4   #.mask (Boolean);
5   #.preorder (list of lists);
```

The three objects `.domain`, `.mask`, and `.preorder` give all the needed information to specify a category of quasi-homologous segments $\mathbf{Seg}(\Omega \,|\, n)$ (as defined in [2, 1]) for some pre-ordered set $(\Omega, \preceq)$ and non-negative integer n. Specifically, the object

- `.domain` contains the integer n defining what is called the *domain* [2, 1] of the segments of $\mathbf{Seg}(\Omega \,|\, n)$, which can informally be identified as the length of the segments;
- `.mask` indicates whether the pre-ordered set $(\Omega, \preceq)$ contains a formal initial object (i.e. a minimum element), which is usually used as an 'ignore' state or a mask;
- `.preorder` contains the down-closure of all the elements of the pre-ordered set Ω. This type of data corresponds to the type of data outputted by the following procedure composition (see section 4.2 and section 4.1)

`_transitive_closure(_read_pre_order(-)[1])`

```
6   def __init__(self,name_of_file,length_of_segments):
7   """ the source code of this constructor can be found in cl_cos.py """
```

For its part, the constructor `.__init__` takes two inputs, namely the name of file that contains a pre-order structure and an integer, and stores

- the integer in the object `.domain`;
- a Boolean value specifying whether the pre-order structure comprises a formal initial object in `.mask`;
- the down-closures of the elements of the pre-order in `.preorder`.

The other methods of the class are very 'categorical', in the sense that they either answers question that are relevant to the category structure of $\mathbf{Seg}(\Omega \,|\, n)$ or provide items that would be directly accessible from its structure.

```
 8   def can_switch_to(self,color1,color2):
 9   """ the source code of this function can be found in cl_cos.py """
10       return (color2 in self.preorder[i])
11   def homset_is_inhabited(self,source,target):
12   """ the source code of this function can be found in cl_cos.py """
13     return flag
14   def topology(self,expression):
15   """ the source code of this function can be found in cl_cos.py """
16     return the_topology
17   def segment(self,expression):
18   """ the source code of this function can be found in cl_cos.py """
19     return the_segment
```

4.4.1. Comparing colors. The method `.can_switch_to` takes two strings that are meant to be names of elements in the pre-ordered set and returns a Boolean value indicating whether the first string is greater than or equal to the second one with respect to the pre-order structure.

For illustration, consider the following pre-order specification.

omega.yml

```
1 !obj:  0 1 2
2 rel:  2 > 1; 0 > 2
```

We can then use the method `.can_switch_to` to verify whether a certain a pre-order relation holds as follows.

```
>>> Seg = CategoryOfSegments("omega.yml",50)
>>> print(Seg.can_switch_to("2","1"))
True
>>> print(Seg.can_switch_to("1","2"))
False
>>> print(Seg.can_switch_to("1","1"))
True
>>> print(Seg.can_switch_to("0","1"))
True
```

4.4.2. Hom-sets. The method `.homset_is_inhabited` takes two `SegmentObject` items (see section 4.3) and returns a Boolean value specifying if there is a morphism from the first one to the second one. Theoretically, this function parses the two segments as if the object `.mask` was set to the value `True`, in which case the topologies of the segments may lack some patches, which are then assumed to have the color of the formal initial object of the pre-ordered set $(\Omega, \preceq)$ (see section 4.4.4).

The following example uses the method `.segment`, which is described in section 4.4.4.

```
>>> Seg = CategoryOfSegments("omega.yml",50)
>>> s1 = Seg.segment([(0,3,10,"1"),(30,6,2,"2")])
>>> s2 = Seg.segment([(0,6,7,"1")])
>>> print(Seg.homset_is_inhabited(s1,s2))
True
>>> s1 = Seg.segment([(0,3,10,"1"),(30,7,2,"2")])
>>> s2 = Seg.segment([(0,6,7,"1")])
>>> print(Seg.homset_is_inhabited(s1,s2))
```

```
False
```

4.4.3. Constructing a topology. The method `.topology` takes a list of 3-tuples and returns a list of 2-tuples. The input list is meant to encode a regular expression that describes the topology of a segment. The 3-tuples contained by the input list specifies

- where a patch starts;
- how long the patch is;
- how many such patches are repeated successively.

Specifically, the input satisfies the following syntax:

```
[(start_position, length_of_patch, number_of_such_patches), (etc.)  ]
```

Also, note that the start positions of the 3-tuples contained in the regular expression need to be given in an increasing order. For instance, a topology of the form

$$(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ)$$

would be encoded by the following lists of 3-tuples:

```
[(0, 3, 2), (6, 4, 1), (10, 3, 4)]
```

However, as is shown, the expression given below is not a valid example.

```
>>> Seg = CategoryOfSegments("omega.yml",22)
>>> expr = [(10, 3, 4), (6, 4, 1), (0, 3, 2)]
>>> s = Seg.topology(expr)
Error:  in CategoryOfSegments.topology:  expression is not well-formed
```

The output of the procedure is the implementation of the regular expression into a topology (a list of 2-tuples). The 2-tuples contained in the output are the start and end positions of each patch. For instance, the output that would be produced for our first example (given above) would be as follows.

```
[(0, 2), (3, 5), (6, 9), (10, 12), (13, 15), (16, 18), (19, 21)]
```

Also, depending on whether the object `.mask` is `True` or `False`, the regular expression either needs to cover the whole topology or only needs to give a few relevant patches whose color might not be that of the initial object of the pre-order structure.

1) If `.mask` contains `True`, then only a few patches can be specified so that the remaining patches of the topology are implicit. For instance, the expression `[(5,3,1)]` can specify either one of the following topologies when `.domain` is equal to `15`.

(4.2)

$$(\circ\circ\circ\circ\circ)(\circ\circ\circ)(\circ\circ\circ\circ\circ\circ\circ) \qquad (\circ)(\circ)(\circ)(\circ)(\circ)(\circ\circ\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)$$

In practice, it is always recommended to associate the implicit part of the topology with the 'mask' color – given by the initial object of $(\Omega, \preceq)$.

```
>>> Seg = CategoryOfSegments("omega.yml",15)
>>> expr = [(5,3,1)]
>>> s = Seg.topology(expr)
>>> print(s)
[(5, 7)]
```

2) If `.mask` contains `False`, then all the patches of the topology must be specified so that the only way to specify the topology given on the left-hand side of (4.2) is to give the following expression.

```
>>> expr = [(0,5,1), (5,3,1), (8,7,1)]
>>> s = Seg.topology(expr)
>>> print(s)
[(0, 4), (5, 7), (8, 14)]
```

4.4.4. Constructing a segment. The method `.segment` takes is very similar to the procedure `.topology`. It takes a list of 4-tuples and returns a `SegmentObject` item. The input list is meant to encode a regular expression that describes the segment. The 4-tuples contained by the input list specifies

- where a patch starts;
- how long the patch is;
- how many such patches are repeated successively;
- the color name (or state) associated with the group of patches.

Specifically, the input satisfies the following syntax:

```
[(start_position, length_of_patch, number_of_such_patches, color), (etc.)  ]
```

Also, note that the start positions of the 4-tuples contained in the regular expression need to be given in an increasing order. For instance, a segment of the form

$$(000)(000)(1111)(222)(222)(222)(222)$$

would be encoded by the following lists of 3-tuples:

```
[(0,3,2,'0'), (6,4,1,'1'), (10,3,4,'2')]
```

However, as is shown, the expression given below is not a valid example.

```
>>> expr = [(10,3,4,'2'), (6,4,1,'1'), (0,3,2,'0')]
>>> s = Seg.segment(expr)
Error:  in CategoryOfSegments.segment:  expression is not well-formed
```

In addition, the color names associated with the patches of the segment must be elements of the pre-order structure specified in the object `.preorder`.

```
>>> Seg = CategoryOfSegments("omega.yml",22)
>>> expr = [(0,3,2,'0'), (6,4,1,'1'), (10,3,4,"red")]
>>> s = Seg.segment(expr)
Error:  in CategoryOfSegments.segment:  color 'red' not found in .preorder
```

The output of the procedure is the implementation of the regular expression into a `SegmentObject` item. For instance, the segment that would be produced for our first example (given above) would be given by the following pair of lists.

```
topology = [(0, 2), (3, 5), (6, 9), (10, 12), (13, 15), (16, 18), (19, 21)]
colors = ['0', '0', '1', '2', '2', '2', '2']
```

Also, depending on whether the object `.mask` is `True` or `False`, the regular expression either needs to cover the whole segment or only needs to give a few relevant patches whose color is not that of the initial object potentially specified in the pre-order structure.

1) If `.mask` contains `True`, then only a few patches can be specified so that the remaining patches of the segment have an implicit topology. For instance, the expression

`[(5,3,1,'1')]` can specify either one of the following segments when `.domain` is equal to `15`.

(4.3)

$$(\circ\circ\circ\circ\circ)(\bullet\bullet\bullet)(\circ\circ\circ\circ\circ\circ\circ) \qquad (\circ)(\circ)(\circ)(\circ)(\circ)(\bullet\bullet\bullet)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)(\circ)$$

Note that, in practice, this polymorphism is never taken into account as it is always associated with the 'mask' color – given by the initial object of $(\Omega, \preceq)$.

```
>>> Seg = CategoryOfSegments("omega.yml",15)
>>> expr = [(5,3,1,'1')]
>>> s = Seg.segment(expr)
>>> print(s.topology)
[(5, 7)]
>>> print(s.colors)
['1']
```

2) If `.mask` contains `False`, then all the patches of the topology must be specified so that the only way to specify the topology given on the left-hand side of (4.3) is to give an expression as follows.

```
>>> expr = [(0,5,1,'0'), (5,3,1,'1'), (8,7,1,'0')]
>>> s = Seg.segment(expr)
>>> print(s.topology)
[(0, 4), (5, 7), (8, 14)]
>>> print(s.colors)
['0', '1', '0']
```

Chapter 5

Presentation of the module `PedigradCategory.py`

5.1. Description of `read_alignment_file`

The function `read_alignment_file` takes the name of a file and an integer and returns a pair of lists.

```
1 def read_alignment_file(name_of_file,reading_mode):
2   """ the source code of this function can be found in raf.py """
3   return (names,alignment)
```

The input file is given in terms of a string and should be as given in the following example, where `This is a text` is a text that does not contain the character '>'.

Align.fa

```
1 >Name of taxon1
2 This is a text
3
4 >Name of taxon2
5 This is a
6 text
```

The second input `reading_mode` specificies whether the text specifically contains a DNA sequence or any regular text. In the former case, the value of `reading_mode` must be 1, which will imply that all lower case letters `a`, `c`, `g` and `t` are read as upper case letters `A`, `C`, `G` and `T`. The global variable `READ_DNA` can be used for this purpose.

```
>>> READ_DNA
1
```

The output `names` is a list of strings containing the names placed after the charater '>' in the input file while the output `alignment` is a list of lists of characters that spell each text displayed after the names saved in `names`.

As can be seen in the example given below, the index of a name in `names` corresponds to the index of its associated text in `alignment`.

```
>>> output = read_alignment_file("Align.fa",READ_DNA)
>>> for i in range(len(output[0])):
...   print(str(i)+":  "+str(output[0][i]))
0:  Name of taxon1
1:  Name of taxon2
>>> for i in range(len(output[1])):
...   print(str(i)+":  "+str(output[1][i]))
0:  ['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'A', ' ', 'T', 'e', 'x', 'T']
1:  ['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'A', 'T', 'e', 'x', 'T']
```

5.2. Description of ID_to_EQ

The function ID_to_EQ takes a string and returns a list of lists, which describes an equivalence relation whose equivalence classes (the internal lists) can be used during the parsing of a sequence alignment to identify certain molecular patches together.

```
1 def ID_to_EQ(name_ID):
2   """ the source code of this function can be found in ite.py """
3     return N21_EQ
4   else:
5     print("Error:  in ID_to_EQ: name_ID is not recognized")
5     exit()
```

The function ID_to_EQ is meant to be used with two sets of global variables. One sets consists of strings that define the valid inputs of ID_to_EQ while the other set consists of the possible outputs of ID_to_EQ, which are lists of lists that are

- either already equipped with specific equivalence classes describing a well-known evolution model (transition mutations, codon translation, etc.)
- or empty so that they can be 'edited' (see below).

We now give three examples of global variables that are associated with pre-determined equivalence classes, the first one being the trivial one, which is to say that it will involve no other identification than the reflexive ones (i.e. the identities).

```
>>> ID_to_EQ(NUCL_ID)
[[]]
```

Using the previous equivalence relation implies that the characters of a sequence alignment are read as-is (see section 5.3 for more details).

The next equivalence relation describes transition mutations (A ↔ G and C ↔ T), which commonly occur on the nucleotides of DNA strands. These are often silent mutations so that one sometimes reasonably wants to identify A with G and C with T.

```
>>> ID_to_EQ(TRAN_ID)
[['A', 'G'], ['C', 'T']]
```

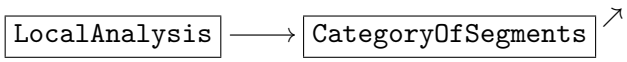
The last equivalence relation contains the equivalence classes induced by the codon table, that is to say those codons coding for the same amino acids.

```
>>> ID_to_EQ(AMIN_ID)
[['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT', 'CTC', 'CTA', 'CTG'], ['TCT',
'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT', 'TAC'], ['CAT', 'CAC'], ['CAA',
'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC', 'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'],
['GTT', 'GTC', 'GTA', 'GTG'], ['ACT', 'ACC', 'ACA', 'ACG'], ['GCT', 'GCC',
'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA', 'AAG'], ['GAT', 'GAC'], ['GAA',
'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC', 'GGA', 'GGG']]
```

The user can also design its own equivalence relation (up to 21) via the (empty) global variables `N01_EQ`, `N02_EQ`, ..., `N21_EQ`, which are associated with the global strings `N01_ID`, `N02_ID`,..., `N21_ID`. The way an equivalence relation can be used with its associated string will be explained in section 5.3.

```
>>> print(N05_ID)
n5
>>> print(ID_to_EQ(N05_ID))
[]
>>> N05_EQ.append(["A","G"])
>>> print(N05_EQ)
[['A', 'G']]
>>> print(ID_to_EQ(N05_ID))
[['A', 'G']]
>>> N04_EQ.extend([["AC","TC"],["CC", "TC"]]))
>>> print(ID_to_EQ(N04_ID))
[['AC', 'TC'], ['CC', 'TC']]
```

5.3. Description of `LocalAnalysis` (subclass)



The class `LocalAnalysis` is a subclass of `CategoryOfSegments` (section 4.4) that possesses three objects, namely

- `.equiv` (list of strings)
- `.base` (list of `SegmentObject` items)

and one method, namely a constructor `.__init__`.

```
1 class LocalAnalysis(CategoryOfSegments):
2   #The objects of the class are:
3   #.equiv (list of strings);
4   #.base (list of SegmentObjects);
5   def __init__(self,analysis_mode,*args):
6   """ the source code of this constructor can be found in cl_la.py """
```

The constructor `.__init__` takes between 4 and 5 arguments and uses them to initialize the objects of the class according to certain pre-determined or personalized parterns. Note that the idea behind the class `LocalAnalysis` is to specify the shape of a local analysis [1] via the two objects `.equiv` and `.base`. This pattern is then used, in section 5.5, to generate what is regarded as a pedigrad [2, 1].

The first argument taken by `.__init__` is always a string. However, the form of its second and third arguments may vary depending on the string contained in the first argument. On

the other hand, the last two arguments always keep the same form, namely a string and an integer, which are given to the constructor of `CategoryOfSegments` to initialize the (super) objects `.domain`, `.mask`, and `.preorder` (see section 4.4)

More specifically, the first argument of `.__init__` is meant to be part of a set of global variables containing strings, which are all displayed below.

```
EXPR_MODE = 'exp'
SEGM_MODE = 'seg'
NUCL_MODE = 'nu'
TRAN_MODE = 'tr'
TRN0_MODE = 'tr0nu'
TRN1_MODE = 'tr1nu'
TRN2_MODE = 'tr2nu'
AMN0_MODE = 'aa0'
AMN1_MODE = 'aa1'
AMN2_MODE = 'aa2'
AMIN_MODE = 'aa'
```

Depending on the string contained in the first argument, the constructor `.__init__` may take four or five arguments. In addition, the second and third arguments may also take different forms.

The procedures `.__init__(NUCL_MODE,−)` and `.__init__(TRAN_MODE,−)` take three arguments whose last two arguments should be as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a string that is the name of an element in the pre-order structure stored in the object `.preorder`.

The following example illustrates the two types of local analysis schemas that are obtained from the global variables `NUCL_MODE` and `NUCL_MODE`. The pre-order structure `omega.yml` used below is that given in section 4.4.1.

```
>>> Loc = LocalAnalysis(NUCL_MODE,'1',"omega.yml",10)
>>> print(Loc.domain)
10
>>> print(Loc.mask)
True
>>> print(Loc.preorder)
[['0', '2', '1'], ['1'], ['2', '1']]
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['1'] ---> nu
segment 1:  [(1, 1)] ['1'] ---> nu
segment 2:  [(2, 2)] ['1'] ---> nu
segment 3:  [(3, 3)] ['1'] ---> nu
segment 4:  [(4, 4)] ['1'] ---> nu
segment 5:  [(5, 5)] ['1'] ---> nu
segment 6:  [(6, 6)] ['1'] ---> nu
segment 7:  [(7, 7)] ['1'] ---> nu
segment 8:  [(8, 8)] ['1'] ---> nu
segment 9:  [(9, 9)] ['1'] ---> nu
```

```
>>> Loc = LocalAnalysis(TRAN_MODE,'2',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] ['2'] ---> tr
segment 1:  [(1, 1)] ['2'] ---> tr
segment 2:  [(2, 2)] ['2'] ---> tr
segment 3:  [(3, 3)] ['2'] ---> tr
segment 4:  [(4, 4)] ['2'] ---> tr
segment 5:  [(5, 5)] ['2'] ---> tr
segment 6:  [(6, 6)] ['2'] ---> tr
segment 7:  [(7, 7)] ['2'] ---> tr
segment 8:  [(8, 8)] ['2'] ---> tr
segment 9:  [(9, 9)] ['2'] ---> tr
```

From the point of view of the function `ID_to_EQ` (see section 5.2), the previous two procedures specify how the images of the local analyses should be 'quotiented' via the strings contained in the object `.equiv`. Specifically, the type of local analysis returned by

`.__init__(NUCL_MODE,−)`

is meant to read the columns of a sequence alignment without identification between the characters while the type of local analysis returned by `.__init__(TRAN_MODE,−)` is meant to identify `A` with `G` and `C` with `T`.

```
>>> Loc = LocalAnalysis(NUCL_MODE,'1',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['1'] ---> [[]]
segment 1:  [(1, 1)] ['1'] ---> [[]]
segment 2:  [(2, 2)] ['1'] ---> [[]]
segment 3:  [(3, 3)] ['1'] ---> [[]]
segment 4:  [(4, 4)] ['1'] ---> [[]]
segment 5:  [(5, 5)] ['1'] ---> [[]]
segment 6:  [(6, 6)] ['1'] ---> [[]]
segment 7:  [(7, 7)] ['1'] ---> [[]]
segment 8:  [(8, 8)] ['1'] ---> [[]]
segment 9:  [(9, 9)] ['1'] ---> [[]]
>>> Loc = LocalAnalysis(TRAN_MODE,'2',"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 1:  [(1, 1)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 2:  [(2, 2)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 3:  [(3, 3)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 4:  [(4, 4)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 5:  [(5, 5)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 6:  [(6, 6)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 7:  [(7, 7)] ['2'] ---> [['A', 'G'], ['C', 'T']]
segment 8:  [(8, 8)] ['2'] ---> [['A', 'G'], ['C', 'T']]
```

```
segment 9:  [(9, 9)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
```

The three procedures

`.__init__(TRN0_MODE,−)`, `.__init__(TRN1_MODE,−)` and `.__init__(TRN2_MODE,−)`

take three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a list of two strings that are the names of elements in the pre-order structure stored in the object `.preorder`.

```
>>> Loc = LocalAnalysis(TRN0_MODE,[‘2’,’1’],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] [‘2’] ---> tr
segment 1:  [(1, 1)] [‘1’] ---> nu
segment 2:  [(2, 2)] [‘1’] ---> nu
segment 3:  [(3, 3)] [‘2’] ---> tr
segment 4:  [(4, 4)] [‘1’] ---> nu
segment 5:  [(5, 5)] [‘1’] ---> nu
segment 6:  [(6, 6)] [‘2’] ---> tr
segment 7:  [(7, 7)] [‘1’] ---> nu
segment 8:  [(8, 8)] [‘1’] ---> nu
segment 9:  [(9, 9)] [‘2’] ---> tr
>>> Loc = LocalAnalysis(TRN2_MODE,[‘2’,‘1’],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 0)] [‘1’] ---> nu
segment 1:  [(1, 1)] [‘1’] ---> nu
segment 2:  [(2, 2)] [‘2’] ---> tr
segment 3:  [(3, 3)] [‘1’] ---> nu
segment 4:  [(4, 4)] [‘1’] ---> nu
segment 5:  [(5, 5)] [‘2’] ---> tr
segment 6:  [(6, 6)] [‘1’] ---> nu
segment 7:  [(7, 7)] [‘1’] ---> nu
segment 8:  [(8, 8)] [‘2’] ---> tr
segment 9:  [(9, 9)] [‘1’] ---> nu
```

From the point of view of the function `ID_to_EQ` (see section 5.2), a procedure of the form

`.__init__(TRNX_MODE,−)`

generates a local analysis that is meant to identify `A` with `G` and `C` with `T` on every column whose position is `X` modulo 3 and that is meant to read the other columns as-is.

```
>>> Loc = LocalAnalysis(TRN0_MODE,[‘2’,’1’],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 1:  [(1, 1)] [‘1’] ---> [[]]
segment 2:  [(2, 2)] [‘1’] ---> [[]]
```

```
segment 3:  [(3, 3)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 4:  [(4, 4)] [‘1’] ---> [[]]
segment 5:  [(5, 5)] [‘1’] ---> [[]]
segment 6:  [(6, 6)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 7:  [(7, 7)] [‘1’] ---> [[]]
segment 8:  [(8, 8)] [‘1’] ---> [[]]
segment 9:  [(9, 9)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
>>> Loc = LocalAnalysis(TRN2_MODE,[‘2’,‘1’],"omega.yml",10)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(0, 0)] [‘1’] ---> [[]]
segment 1:  [(1, 1)] [‘1’] ---> [[]]
segment 2:  [(2, 2)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 3:  [(3, 3)] [‘1’] ---> [[]]
segment 4:  [(4, 4)] [‘1’] ---> [[]]
segment 5:  [(5, 5)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 6:  [(6, 6)] [‘1’] ---> [[]]
segment 7:  [(7, 7)] [‘1’] ---> [[]]
segment 8:  [(8, 8)] [‘2’] ---> [[‘A’, ‘G’], [‘C’, ‘T’]]
segment 9:  [(9, 9)] [‘1’] ---> [[]]
```

The three procedures

`.__init__(AMN0_MODE,−)`, `.__init__(AMN1_MODE,−)` and `.__init__(AMN2_MODE,−)`

take three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a string that is the name of an element in the pre-order structure stored in the object `.preorder`.

```
>>> Loc = LocalAnalysis(AMN0_MODE,‘1’,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 2)] [‘1’] ---> aa
segment 1:  [(3, 5)] [‘1’] ---> aa
segment 2:  [(6, 8)] [‘1’] ---> aa
segment 3:  [(9, 11)] [‘1’] ---> aa
segment 4:  [(12, 14)] [‘1’] ---> aa
segment 5:  [(15, 17)] [‘1’] ---> aa
segment 6:  [(18, 20)] [‘1’] ---> aa
>>> Loc = LocalAnalysis(AMN1_MODE,‘1’,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(1, 3)] [‘1’] ---> aa
segment 1:  [(4, 6)] [‘1’] ---> aa
segment 2:  [(7, 9)] [‘1’] ---> aa
segment 3:  [(10, 12)] [‘1’] ---> aa
segment 4:  [(13, 15)] [‘1’] ---> aa
segment 5:  [(16, 18)] [‘1’] ---> aa
```

```
>>> Loc = LocalAnalysis(AMN2_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(2, 4)] ['1'] ---> aa
segment 1:  [(5, 7)] ['1'] ---> aa
segment 2:  [(8, 10)] ['1'] ---> aa
segment 3:  [(11, 13)] ['1'] ---> aa
segment 4:  [(14, 16)] ['1'] ---> aa
segment 5:  [(17, 19)] ['1'] ---> aa
```

From the point of view of the function ID_to_EQ (see section 5.2), a procedure of the form

```
.__init__(AMNX_MODE,—)
```

generates a local analysis that reads the codons of a sequence alignment with the assumption that the codon topology starts at position X and identifies them according to the codon translation table.

```
>>> Loc = LocalAnalysis(AMN2_MODE,'1',"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(2, 4)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 1:  [(5, 7)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 2:  [(8, 10)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
segment 3:  [(11, 13)] ['1'] ---> [['TTT', 'TTC'], ['TTA', 'TTG'], ['CTT',
'CTC', 'CTA', 'CTG'], ['TCT', 'TCC', 'TCA', 'TCG', 'AGT', 'AGC'], ['TAT',
'TAC'], ['CAT', 'CAC'], ['CAA', 'CAG'], ['TGT', 'TGC'], ['CGT', 'CGC',
'CGA', 'CGG'], ['ATT', 'ATC', 'ATA'], ['GTT', 'GTC', 'GTA', 'GTG'], ['ACT',
'ACC', 'ACA', 'ACG'], ['GCT', 'GCC', 'GCA', 'GCG'], ['AAT', 'AAC'], ['AAA',
'AAG'], ['GAT', 'GAC'], ['GAA', 'GAG'], ['AGA', 'AGG'], ['GGT', 'GGC',
'GGA', 'GGG']]
```

```
segment 4:  [(14, 16)] [‘1’] ---> [[‘TTT’, ‘TTC’], [‘TTA’, ‘TTG’], [‘CTT’,
‘CTC’, ‘CTA’, ‘CTG’], [‘TCT’, ‘TCC’, ‘TCA’, ‘TCG’, ‘AGT’, ‘AGC’], [‘TAT’,
‘TAC’], [‘CAT’, ‘CAC’], [‘CAA’, ‘CAG’], [‘TGT’, ‘TGC’], [‘CGT’, ‘CGC’,
‘CGA’, ‘CGG’], [‘ATT’, ‘ATC’, ‘ATA’], [‘GTT’, ‘GTC’, ‘GTA’, ‘GTG’], [‘ACT’,
‘ACC’, ‘ACA’, ‘ACG’], [‘GCT’, ‘GCC’, ‘GCA’, ‘GCG’], [‘AAT’, ‘AAC’], [‘AAA’,
‘AAG’], [‘GAT’, ‘GAC’], [‘GAA’, ‘GAG’], [‘AGA’, ‘AGG’], [‘GGT’, ‘GGC’,
‘GGA’, ‘GGG’]]
segment 5:  [(17, 19)] [‘1’] ---> [[‘TTT’, ‘TTC’], [‘TTA’, ‘TTG’], [‘CTT’,
‘CTC’, ‘CTA’, ‘CTG’], [‘TCT’, ‘TCC’, ‘TCA’, ‘TCG’, ‘AGT’, ‘AGC’], [‘TAT’,
‘TAC’], [‘CAT’, ‘CAC’], [‘CAA’, ‘CAG’], [‘TGT’, ‘TGC’], [‘CGT’, ‘CGC’,
‘CGA’, ‘CGG’], [‘ATT’, ‘ATC’, ‘ATA’], [‘GTT’, ‘GTC’, ‘GTA’, ‘GTG’], [‘ACT’,
‘ACC’, ‘ACA’, ‘ACG’], [‘GCT’, ‘GCC’, ‘GCA’, ‘GCG’], [‘AAT’, ‘AAC’], [‘AAA’,
‘AAG’], [‘GAT’, ‘GAC’], [‘GAA’, ‘GAG’], [‘AGA’, ‘AGG’], [‘GGT’, ‘GGC’,
‘GGA’, ‘GGG’]]
```

The procedure `.__init__(AMIN_MODE,−)`, takes three arguments whose last two arguments are as described above, namely a string and an integer meant to be fed to the method `.__init__` of `CategoryOfSegments`. For its part, the first argument should be a list of three strings that are the names of elements in the pre-order structure stored in the object `.preorder`. From the point of view of the function `ID_to_EQ` (see section 5.2), the procedure

`.__init__(AMIN_MODE,−)`

generates a local analysis that reads all the codons of the sequence and identifies them according to the codon translation table.

```
>>> Loc =LocalAnalysis(AMIN_MODE,[‘0’,‘1’,‘2’],"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(0, 2)] [‘0’] ---> aa
segment 1:  [(3, 5)] [‘0’] ---> aa
segment 2:  [(6, 8)] [‘0’] ---> aa
segment 3:  [(9, 11)] [‘0’] ---> aa
segment 4:  [(12, 14)] [‘0’] ---> aa
segment 5:  [(15, 17)] [‘0’] ---> aa
segment 6:  [(18, 20)] [‘0’] ---> aa
segment 7:  [(1, 3)] [‘1’] ---> aa
segment 8:  [(4, 6)] [‘1’] ---> aa
segment 9:  [(7, 9)] [‘1’] ---> aa
segment 10:  [(10, 12)] [‘1’] ---> aa
segment 11:  [(13, 15)] [‘1’] ---> aa
segment 12:  [(16, 18)] [‘1’] ---> aa
segment 13:  [(2, 4)] [‘2’] ---> aa
segment 14:  [(5, 7)] [‘2’] ---> aa
segment 15:  [(8, 10)] [‘2’] ---> aa
segment 16:  [(11, 13)] [‘2’] ---> aa
segment 17:  [(14, 16)] [‘2’] ---> aa
segment 18:  [(17, 19)] [‘2’] ---> aa
```

Finally the user can design their own local analysis by using the procedures

`.__init__(EXPR_MODE,−)` and `.__init__(SEGM_MODE,−)`,

which take four more arguments as described below:

- the first argument of .__init__(EXPR_MODE,−) should be a list of regular expressions specifying SegmentObject items (section 4.3). This list is then turned into a list of SegmentObject items, which is to be stored in the object .base;

- the first argument of .__init__(SEGM_MODE,−) should be a list of SegmentObject items (section 4.3), which is to be stored in the object .base;

- the second argument should be a list of strings, which is to be stored in the object .equiv. More specifically, the strings should be among those contained in the global variables NUCL_ID, TRAN_ID, AMIN_ID, N01_ID, N02_ID, ..., and N21_ID;

- and, as mentioned many times above, the last two arguments should be a string and an integer, which are to be passed to the method .__init__ of CategoryOfSegments (see section 4.4).

Below, we show two examples of how the previous two procedures can be used. Note how theses examples uses the (empty) global variables N01_EQ and N02_EQ (see section 5.2).

```
>>> N01_EQ.append(["A","G"])
>>> N02_EQ.extend([["AC","TC"],["CC", "TC"]])
>>> base = [[(5,1,1,'1')], [(6,2,1,'2')], [(8,1,1,'1')], [(9,2,2,'2')]]
>>> equiv = [N01_ID, N02_ID, N01_ID, N02_ID, N02_ID]
>>> Loc = LocalAnalysis(EXPR_MODE,base,equiv,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
segment 0:  [(5, 5)] ['1'] ---> n1
segment 1:  [(6, 7)] ['2'] ---> n2
segment 2:  [(8, 8)] ['1'] ---> n1
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> n2
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(5, 5)] ['1'] ---> [['A', 'G']]
segment 1:  [(6, 7)] ['2'] ---> [['AC', 'TC'], ['CC', 'TC']]
segment 2:  [(8, 8)] ['1'] ---> [['A', 'G']]
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> [['AC', 'TC'], ['CC', 'TC']]
```

Alternatively, the procedure .__init__(SEGM_MODE,−) would require to have a pre-defined environment in which SegmentObject items can be defined. Note the difference between the previous lines of codes and the following ones, which use the class CategoryOfSegments.

```
>>> N01_EQ.append(["A","G"])
>>> N02_EQ.extend([["AC","TC"],["CC", "TC"]])
>>> Seg = CategoryOfSegments("omega.yml",21)
>>> base = [Seg.segment([(5,1,1,'1')]), Seg.segment([(6,2,1,'2')]),
    Seg.segment([(8,1,1,'1')]), Seg.segment([(9,2,2,'2')])]
>>> equiv = [N01_ID, N02_ID, N01_ID, N02_ID, N02_ID]
>>> Loc = LocalAnalysis(SEGM_MODE,base,equiv,"omega.yml",21)
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(Loc.equiv[i]))
```

```
segment 0:  [(5, 5)] ['1'] ---> n1
segment 1:  [(6, 7)] ['2'] ---> n2
segment 2:  [(8, 8)] ['1'] ---> n1
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> n2
>>> for i in range(len(Loc.base)):
...   print("segment " + str(i) + ":  " + str(Loc.base[i].topology) + " " +
    str(Loc.base[i].colors) + " ---> " + str(ID_to_EQ(Loc.equiv[i])))
segment 0:  [(5, 5)] ['1'] ---> [['A', 'G']]
segment 1:  [(6, 7)] ['2'] ---> [['AC', 'TC'], ['CC', 'TC']]
segment 2:  [(8, 8)] ['1'] ---> [['A', 'G']]
segment 3:  [(9, 10), (11, 12)] ['2', '2'] ---> [['AC', 'TC'], ['CC', 'TC']]
```

5.4. Description of column_is_trivial

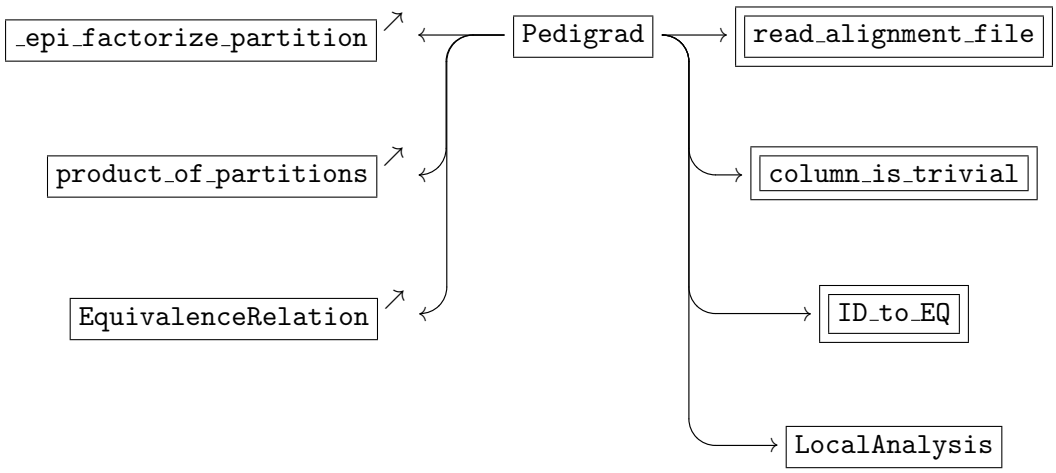
The function column_is_trivial takes two lists of elements and returns True if, apart from the elements in the second input list, the first input list contains copies of a unique element; returns False if, apart from the elements in the second input list, the first input list contains at least two different elements.

```
1 def column_is_trivial(column,exceptions):
2   """ the source code of this function can be found in cit.py """
3   return flag
```

As will be seen later in other functions, there are often special characters that need to be handled differently from the other characters. For instance, the character '.' is often used in alignments to mean that a character is actually missing. These special characters sometimes need to be ignored in the analysis.

```
>>> p = ['e','e','e','e','e']
>>> print(column_is_trivial(p,[]))
True
>>> p = ['e','e','a','a','e']
>>> print(column_is_trivial(p,[]))
False
>>> p = [0,'.',0,0,'.',0,1,0]
>>> print(column_is_trivial(p,['.']))
False
>>> print(column_is_trivial(p,['.',1]))
True
```

5.5. Description of Pedigrad (subclass)



The class Pedigrad is a subclass of LocalAnalysis (section 5.3) that possesses two objects, namely

- .local (list of lists)
- .taxa (list)

and two methods, namely

- .__init__ (constructor)
- .partition
- .select
- .common

The constructor .__init__ takes from 5 to 6 arguments, which are meant to be used with the function read_alignment_file and the constructor of LocalAnalysis. More specifically, the constructor of Pedigrad usually takes

- the name of a file and an integer, given to the function read_alignment_file;
- a list of arguments, given to the constructor of LocalAnalysis.

```
1 class Pedigrad(LocalAnalysis):
2   #The objects of the class are:
3   #.local (list);
4   #.taxa (list).
5   def __init__(self,name_of_file,reading_mode,*args):
6   """ the source code of this constructor can be found in cl_ped.py """
```

The list of arguments *args (shown above) usually constains those arguments that would be given to the constructor of LocalAnalysis (see .__init__ in section 5.3), except for the last argument, which does not need to be given if

- the name of the file is the name of an existing file;
- the name of the file is empty (see the examples given later).

For illustration, we will use the preorder structure omega.yml given in section 4.4.1 and the following alignment file.

Align.fa

```
 1 > A
 2 aaaaaaaaaf
 3 > B
 4 bbbbbbbbbf
 5 > C
 6 ccccccccf
 7 > D
 8 ddddddddf
 9 > E
10 eeeeeeeeef
```

The following lines of code illustrate the syntax with which an instance of Pedigrad can be constructed and compares it to that used to construct an instance of LocalAnalysis.

```
>>> P = Pedigrad("align.fa",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml")
>>> L = LocalAnalysis(TRN2_MODE,['2','1'],"omega.yml",500)
>>> P = Pedigrad("align.fa",not(READ_DNA),AMNO_MODE,'1',"omega.yml")
>>> L = LocalAnalysis(AMNO_MODE,'1',"omega.yml",500)
>>> P = Pedigrad("",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml")
TypeError:  __init__() takes exactly 3 arguments (2 given)
>>> P = Pedigrad("",not(READ_DNA),TRN2_MODE,['2','1'],"omega.yml",500)
```

If the name of the file that is passed to the contructor, in its first argument, is not empty, then the first two arguments are passed to the procedure `read_alignment_file(−,−)` and the constructor proceeds to the initialization of the objects `.local`, `.taxa` and `.base` as follows:

- it stores the first output of `read_alignment_file` in the object `.taxa`;
- it appends the remaining lists of arguments with the length of the second output of `read_alignment_file` and gives the resulting tuples of arguments to the constructor of `LocalAnalysis` (see below).

```
super(Pedigrad, self).__init__(*args+(len(alignment[0]),))
```

Once the object `.base` of the super class has been (pre-)initialized by this process, the `SegmentObject` items that it contains are used to parse the sequence alignment. The parsed data is then stored in the object `.local` as follows:

- ▷ For every patch `(x,y)` occurring in the topology of i-th segment contained `.base`, the characters appearing from position `x` to position `y`, in the sequence alignment, are collected for each line of the sequence alignment (i.e. each list of characters contained in the second output of `read_alignment_file`) and put together to form a list of strings;
- ▷ This list of strings is then relabeled with respect to the indices of the lists of equivalence classes that should have been among the arguments meant to be given to the constructor of `LocalAnalysis`;
- ▷ If the relabeled list of strings is not trivial according to the procedure

  `column_is_trivial(−,[])`,

  then it is stored at position i in the object `.local`;
- ▷ If the lists of strings happens to be trivial, then it is not stored and the associated segment is removed from the object `.base` (thus shifting the indexing).

```
>>> print(NUCL_EQ)
[[]]
>>> P = Pedigrad("align.fa",not(READ_DNA),NUCL_MODE,'2',"omega.yml")
>>> print(P.taxa)
[' A', ' B', ' C', ' D', ' E']
>>> print(P.domain,len(P.base),len(P.local))
(10, 9, 9)
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 1:  [(1, 1)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 2:  [(2, 2)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 3:  [(3, 3)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 4:  [(4, 4)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 5:  [(5, 5)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 6:  [(6, 6)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 7:  [(7, 7)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
segment 8:  [(8, 8)], ['2'] ---> ['a', 'b', 'c', 'd', 'e']
```

When the `SegmentObjects` items of the base only contain patches of length 1 and the equivalence classes are empty (as above), then the list of lists stored in `.local` corresponds to the transpose of the sequence alignment (when seen as a matrix), up to removal of those columns (in the alignment) that only contain the same character.

Below, we give several other examples, some of which present bases whose `SegmentObject` items contain patches of length greater than 1.

Our first example illustrates how the lists of strings contained in the object `.local` are relabeled when the equivalence class contained in`NUCL_EQ` is replaced with non-trivial ones.

```
>>> NUCL_EQ.remove([])
>>> NUCL_EQ.extend([['e'],['d'],['c'],['b'],['a']])
>>> print(NUCL_EQ)
[['e'], ['d'], ['c'], ['b'], ['a']]
>>> P = Pedigrad("align.fa",not(READ_DNA),NUCL_MODE,'2',"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], ['2'] ---> [4, 3, 2, 1, 0]
segment 1:  [(1, 1)], ['2'] ---> [4, 3, 2, 1, 0]
segment 2:  [(2, 2)], ['2'] ---> [4, 3, 2, 1, 0]
segment 3:  [(3, 3)], ['2'] ---> [4, 3, 2, 1, 0]
segment 4:  [(4, 4)], ['2'] ---> [4, 3, 2, 1, 0]
segment 5:  [(5, 5)], ['2'] ---> [4, 3, 2, 1, 0]
segment 6:  [(6, 6)], ['2'] ---> [4, 3, 2, 1, 0]
segment 7:  [(7, 7)], ['2'] ---> [4, 3, 2, 1, 0]
segment 8:  [(8, 8)], ['2'] ---> [4, 3, 2, 1, 0]
```

This second example is similar to the previous one, but the relabelling only occurs where the lists of strings conained in the object `.local` are detected. Note that the following example uses the modified version `NUCL_EQ` designed in the previous example.

```
>>> print(NUCL_EQ)
[[‘e’], [‘d’], [‘c’], [‘b’], [‘a’]]
>>> print(TRAN_EQ)
[[‘A’, ‘G’], [‘C’, ‘T’]]
>>> P = Pedigrad("align.fa",not(READ_DNA),TRN2_MODE,[‘2’,‘1’],"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], [‘1’] ---> [4, 3, 2, 1, 0]
segment 1:  [(1, 1)], [‘1’] ---> [4, 3, 2, 1, 0]
segment 2:  [(2, 2)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 3:  [(3, 3)], [‘1’] ---> [4, 3, 2, 1, 0]
segment 4:  [(4, 4)], [‘1’] ---> [4, 3, 2, 1, 0]
segment 5:  [(5, 5)], [‘2’] ---> [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
segment 6:  [(6, 6)], [‘1’] ---> [4, 3, 2, 1, 0]
segment 7:  [(7, 7)], [‘1’] ---> [4, 3, 2, 1, 0]
```

We finish with the following example, in which the local analysis is designed by the user.

```
>>> N01_EQ.extend([[‘aaa’,‘eee’],[‘bbb’]])
>>> alignment = read_alignment_file("align.fa",not(READ_DNA))
>>> domain = len(alignment[1][0])
>>> base = list()
>>> equiv = list()
>>> for i in range(domain/3):
...   base.append([(3*i,3,1,‘1’)])
...   equiv.append(N01_ID)
>>> P = Pedigrad("align.fa",not(READ_DNA),EXPR_MODE,base,equiv,"omega.yml")
>>> print(P.domain,len(P.base),len(P.local))
(10, 3, 3)
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 2)], [‘1’] ---> [0, 1, ‘ccc’, ‘ddd’, 0]
segment 1:  [(3, 5)], [‘1’] ---> [0, 1, ‘ccc’, ‘ddd’, 0]
segment 2:  [(6, 8)], [‘1’] ---> [0, 1, ‘ccc’, ‘ddd’, 0]
```

In the case where the name of the file that is passed to the contructor, in its first argument, is empty, then the objects `.local` and `.taxa` are initialized with empty lists while the other objects `.equiv`, `.base`, `.domain`, `.mask` and `.preorder` are initialized via the constructor of the class `LocalAnalysis`.

Let us now focus on the other methods of the class, which recover usual or canonical operations on pedigrads.

```
7   def partition(self,*args):
8   """ the source code of this constructor can be found in cl_ped.py """
9       return the_image
10   def select(self,equivalence,*args):
11   """ the source code of this constructor can be found in cl_ped.py """
12     return new_pedigrad
```

```
13   def common(selftree_base,pulling_condition):
14   """ the source code of this constructor can be found in cl_ped.py """
15     return the_common
```

5.5.1. Partition. The method `.partition` takes

- either no argument;
- or the regular expression of `SegmentObject` item and a string (specifically `EXPR_MODE`);
- or a `SegmentObject` item and a string (specifically `SEGM_MODE`).

If no argument is given, then the procedure returns a terminal partition. Otherwise, the procedure returns the image of the right Kan extension (see [1] for more details) of the underlying local analysis on the input segment.

$$\mathbb{R}P(\tau) = \prod_{v \in L} \prod_{\mathbf{Seg}(\Omega \,|\, n)(\tau, v)} P(v)$$

This image is obtained from the product of those partitions, in the object `.local`, whose indices correspond to the indices of those segments in the object `.base` toward which there is a morphism of segment from the input `SegmentObject` item.

For illustration, let us change the sequence alignment to the following one.

Align.fa

```
 1 > A
 2 CCCAGTTAG
 3 > B
 4 CCGATATAA
 5 > C
 6 GGCTTATAG
 7 > D
 8 GGCGTATAG
 9 > E
10 ACCATATAA
```

For simplicity, we shall suppose that the global variable is as initially provided by the library `NUCL_EQ` (empty) and we shall consider the following pedigrad.

```
>>> print(NUCL_EQ)
[[]]
>>> P = Pedigrad("align.fa",not(READ_DNA),NUCL_MODE,'2',"omega.yml")
>>> for i in range(len(P.base)):
...   print("segment " + str(i) + ":  " + str(P.base[i].topology) + ",
    "+str(P.base[i].colors) + " ---> " + str(P.local[i]))
segment 0:  [(0, 0)], ['2'] ---> ['C', 'C', 'G', 'G', 'A']
segment 1:  [(1, 1)], ['2'] ---> ['C', 'C', 'G', 'G', 'C']
segment 2:  [(2, 2)], ['2'] ---> ['C', 'G', 'C', 'C', 'C']
segment 3:  [(3, 3)], ['2'] ---> ['A', 'A', 'T', 'G', 'A']
segment 4:  [(4, 4)], ['2'] ---> ['G', 'T', 'T', 'T', 'T']
segment 5:  [(5, 5)], ['2'] ---> ['T', 'A', 'A', 'A', 'A']
segment 6:  [(8, 8)], ['2'] ---> ['G', 'A', 'G', 'G', 'A']
```

First, recall that if no argument is given to the procedure `.partition`, then the procedure returns the terminal partition, whose (unique) value is chosen to be 0.

```
>>> print(P.partition())
[0, 0, 0, 0, 0]
```

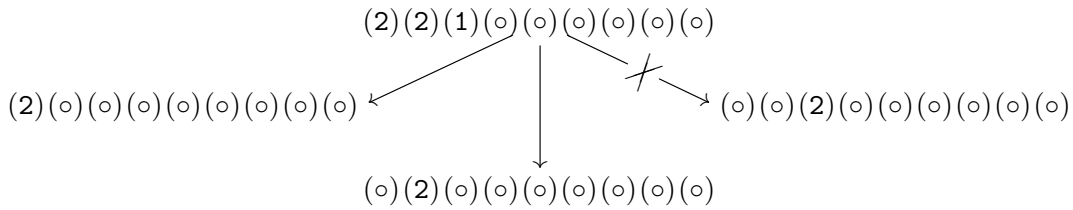
Let us now look at the outputs of the procedure `partition` when the regular expression of a segment is given as an input. For instance, suppose that one wants to compute the image of the following segment via the previously specified pedigrad.

(5.1) (2)(2)(1)(∘)(∘)(∘)(∘)(∘)(∘)

This would be encoded by the following specification.

```
>>> print(P.partition([(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'1')],EXPR_MODE))
[0, 0, 1, 1, 2]
```

Recall that the earlier output is the product of those partitions, in `.local`, whose indices correspond to the indices of those segments, in `.base`, toward which there is a morphism of segment from the input `SegmentObject` item.



This calculation is made more explicit in the following lines of code, in which the segments of the base are given by the regular expressions `[(0,1,1,'2')]`, `[(1,1,1,'2')]`, and `[(2,1,1,'2')]`.

```
>>> s1 = P.segment([[(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'1')]])
>>> s2 = P.segment([(0,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s2))
True
>>> s3 = P.segment([(1,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s3))
True
>>> s4 = P.segment([(2,1,1,'2')]))
>>> print(P.homset_is_inhabited(s1,s4))
False
>>> p1 = P.partition(s1,SEGM_MODE)
>>> p2 = P.partition(s2,SEGM_MODE)
>>> print(product_of_partitions(p1,p2))
[0, 0, 1, 1, 2]
```

For the sake of comparison, we can now compute the image of the following segment.

(5.2) (2)(2)(2)(∘)(∘)(∘)(∘)(∘)(∘)

```
>>> print(P.partition([(0,1,1,'2'),(1,1,1,'2'),(2,1,1,'2')],EXPR_MODE))
[0, 1, 2, 2, 3]
```

The difference between the image of segment (5.1) and that of segment (5.2) comes from non-triviality if the following image.

```
>>> print(P.partition([(2,1,1,'2')],EXPR_MODE))
[0, 1, 0, 0, 0]
```

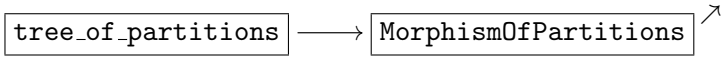
5.5.2. Select.

5.5.3. Common.

Chapter 6

Presentation of the module AsciiTree.py

6.1. Description of tree_of_partitions



The function `tree_of_partitions` takes a list of partitions that can successively be related by morphisms of partitions and returns the actual lists of morphisms of partitions between these.

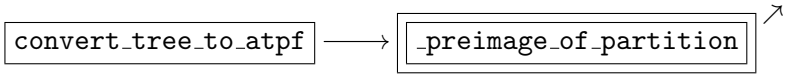
```
1 def tree_of_partitions(partitions):
2   """ the source code of this function can be found in top.py """
3   return the_tree
```

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]
```

6.2. Description of convert_tree_to_atpf

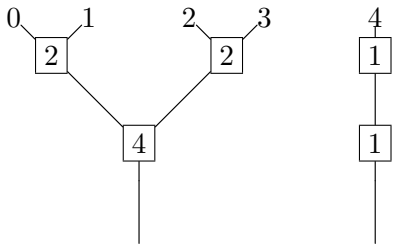


The function convert_tree_to_atpf takes a list of morphisms of partitions (as returned by the procedure tree_of_partitions) and converts it into its associated ascii tree preformat (abbrev. atpf), which is defined as any ATPF term that be can constructed from the following double grammar rules.

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 behind such 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 an 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 one could call the *depth* of the bracketing structure in 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 6.3 and section 6.4), the main task of the procedure convert_tree_to_atpf is to compute the atpf associated with the input list of composable morphisms of partitions by considering the successive preimages of each morphisms of partitions 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 induces an obvious sequence of morphisms of partitions that can be constructed by the procedure tree_of_partitions (see section 6.1).

Since the lengths of these partitions is 6, the atpf will induce a higher level bracketing of the elemens of the list [0,1,2,3,4,5]. To start with, one considers the preimage of the last partition c whose preimage is given by the following list of lists:

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

To start constructing the atpf, we follow the recursive definition of the grammar of atpfs (given above) by taking 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 given by the following list, in which the red numbers are the weight of the list terms (see grammar rule for Leaf).

```
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 arrow encoding the morphism of partitions c → b. Specifically, the arrow of the morphism c → b is as follows (when c and b are relabeled 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
```

Since b has three elements in its image, the morphism f (c → b) 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, we now need to replace

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

so that the 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` $\rightarrow$ `a` and the earlier list 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])])])]
```

6.3. Description of `convert_atpf_to_atf`

The function `convert_atpf_to_atf` takes an atpf and its depth (see section 6.2) and returns the associated ascii tree format, 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

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

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 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` (see section 6.4) to know when it needs to stop printing the horizontal level of the tree, as shown below with the red underscores symbols sticking out toward the right.

```
|                 |
|_________________|
|             |   |
|_____________|   |          atpf_weight = 4
|   |   |     |   |          atf_weight = 4 - 1 = 3
A   C   D...E   F   B
```

6.4. Description of print_atf

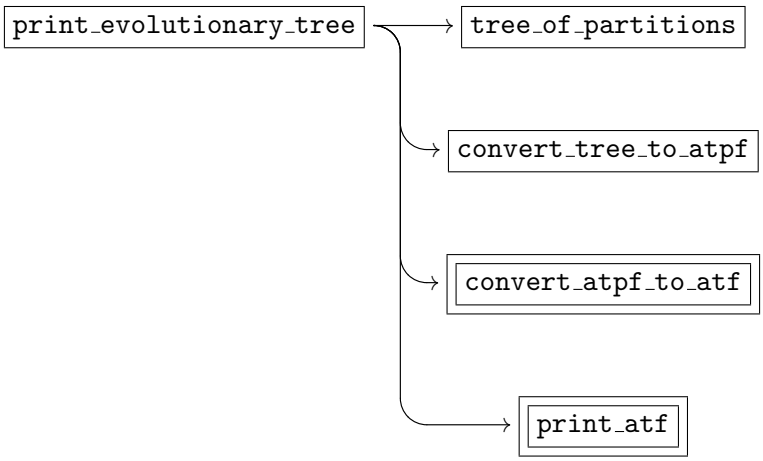
The function print_atf takes an atf and its depth and prints the ascii tree associated with the atf on the standard output.

```
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
```

6.5. Description of print_evolutionary_tree



The function print_evolutionary_tree takes a list of partitions between which a sequence of composable morphisms exists 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])
```

Bibliography

[1] Rémy Tuyéras, *Categorical approach to tree inference*, arXiv:????
[2] —, *Category Theory for Genetics*, arXiv:1708.05255