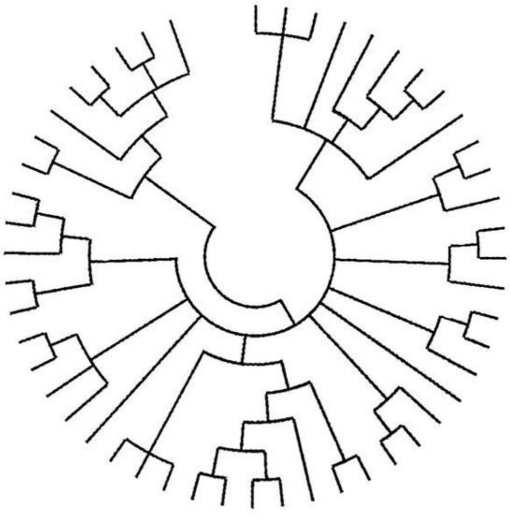
DOCUMENTATION FOR THE PYTHON LIBRARY PEDIGRAD.PY

Rémy Tuyéras
Department of Mathematics
Massachusetts Institute of Technology

Version 1.0



Typeset in LaTeX 2_ε.

Contents

Chapter 1

Introduction

1.1. About pedigrads and Pedigrad.py

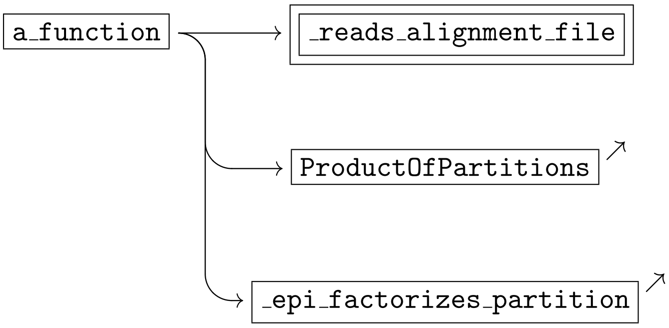
Pedigrads are mathematical tools that were initially created to model various mechanisms of genetics (see [2]). Mathematically, pedigrads are defined as functors sending the cones of a certain type of limit sketch to cones in a given category of values. Pedigrads can encode different aspects of biology 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)
- `PedigradCategory.py` (see Chapter 4)
- `AsciiTree.py` (see Chapter 5)

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.

```
class Pedigrad:
  """
  This is a comment
  """
  def __init__(self,alignment):  # comment:  constructor of the class
    """
    Another comment about the code
    """
```

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.

Chapter 2

Tutorial

To be written

Chapter 3

Presentation of the module `PartitionCategory.py`

3.1. Description of `_image_of_partition`

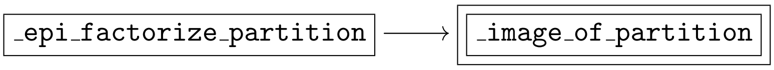
The function `_image_of_partition` takes a list of elements and returns the list of its elements without repetition in the order in which they can be accessed first through the input list.

```python
def _image_of_partition(partition):
  """ the source code of this function can be found in iop.py """
  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.

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

Even though a list already encodes an epimorphism, the function `_image_of_partition` returns a canonical *choice* of epimorphism.

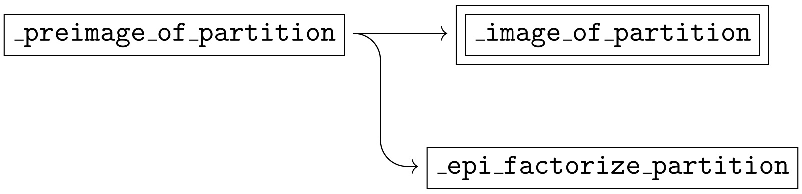
$$\begin{tikzcd}
S \arrow[r, "e(f)"'] \arrow[rr, bend left, "f"] & \mathrm{Im}(f) \arrow[r, "\cong"'] & K
\end{tikzcd}$$

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.

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

For the point of view of partitions, the returned list the_preimage is the preimage of the underlying epimorphism of the input partition $f : S \to K$, where the preimage 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 point of view, 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 `_quotient_of_preimage`

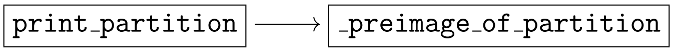
The function `_quotient_of_preimage` takes a list of lists of indices (meant to be the preimage of a partition) and returns a partition (i.e the list) whose preimage is that given in the input.

```python
def _quotient_of_preimage(preimage):
  """ the source code of this function can be found in qop.py """
  return the_quotient
```

An interesting feature of the procedure `_quotient_of_preimage(-)` is that the composition `_quotient_of_preimage(_preimage_of_partition(-))` is equal to the procedure `_epi_factorize_partition(-)`

```
>>> l = [1,2,4,5,1,2,3,2,2,1,3]
>>> p = _preimage_of_partition(l)
>>> print(_quotient_of_preimage(p))
[0, 1, 2, 3, 0, 1, 4, 1, 1, 0, 4]
>>> print(_epi_factorize_partition(l))
[0, 1, 2, 3, 0, 1, 4, 1, 1, 0, 4]
```

3.5. Description of `print_partition`

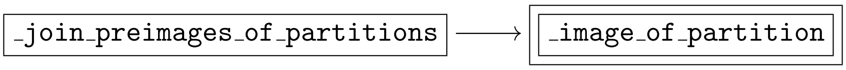


The function `print_partition` is a debug function that takes a list of elements and prints its preimage on the standard output (i.e. the console).

```python
def print_partition(partition):
  print(_preimage_of_partition(partition))
```

See section 3.3 for examples.

3.6. Description of `_join_preimages_of_partitions`



The function `_join_preimages_of_partitions` takes a pair of lists of disjoint lists of indices (the indices should not be repeated and should range from 0 to a fixed positive integer) and returns the list of the maximal unions of internal lists that intersect within the concatenation of the two input lists (see below).

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

In practice, the two input lists `preimage1` and `preimage2` should/would be obtained as outputs of the procedure

`_preimage_of_partition(−,−)`

for two 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(−)`

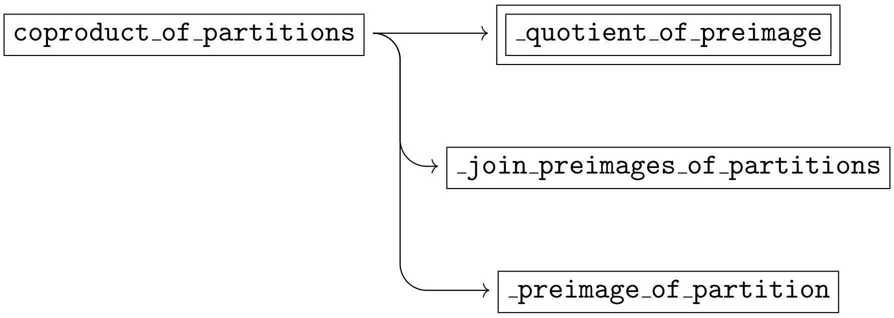
does not necessarily presents its output in the same way as the procedure

`_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]]
```

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.

```
def coproduct_of_partitions(partition1,partition2):
  if len(partition1) == len(partition2):
    #Returns one of the possible constructions of the coproduct
    #of two partitions
    return _quotient_of_preimage(
_join_preimages_of_partitions(
_preimage_of_partition(partition1),
_preimage_of_partition(partition2)))
  else:
    print("Error:  in coproduct_of_partitions:  lengths do not match.")
    exit()
```

Contrary to the type of property that was pointed out in section 3.4, 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 3.6).

```
>>> 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`

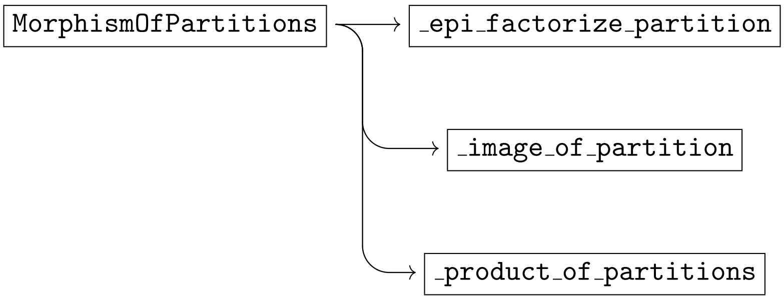
The function `_product_of_partitions` takes two lists and returns the list of pairs of element with the same index in the two lists.

```
def _product_of_partitions(partition1,partition2):
  """ the source code of this function can be found in ptop.py """
  return the_product
```

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

```
>>> _product_of_partitions([1,2,3],[‘a’,‘b’,‘c’])
[(1, ‘a’), (2, ‘b’), (3, ‘c’)]
>>> _product_of_partitions([1,2,3,4],[‘a’,‘b’,‘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.

```
class MorphismOfPartitions:
  #The objects of the class are:
  #.arrow (list);
  #.source (list);
  #.target (list).
  def __init__(self,source,target):
  """ 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 computed as the image of the product 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 product is as follows:

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

The image of the product 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 product of p1 and p2 is as follows:

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

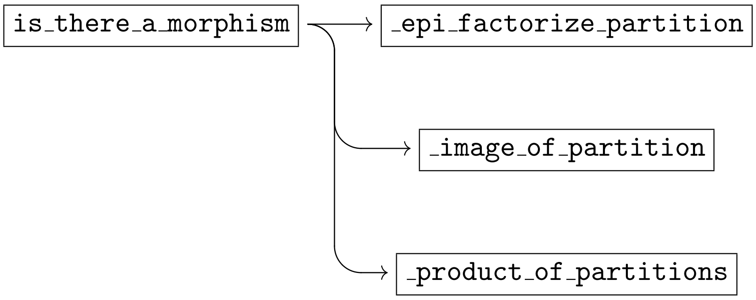
As can be seen, the argument 3 is 'mapped' to two different images, namely 3 and 6. In this case, the constructor .__init__ exits the program with an error message.

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 is_there_a_morphism



The function is_there_a_morphism 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.

```
def is_there_a_morphism(source,target):
  """ the source code of this function can be found in itam.py """
  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]
>>> is_there_a_morphism(p1,p2)
1
>>> p2 = [0,1,2,3,6,3,1]
>>> is_there_a_morphism(p1,p2)
0
```

3.11. Description of SpanOfPartitions (class)

The class SpanOfPartitions possesses three objects, namely

- .peak (list)
- .left (MorphismOfPartitions)
- .right (MorphismOfPartitions)

and a constructor .__init__. The consructor .__init__ takes two MorphismOfPartitions (see section 3.9) that should have the same .source object and stores

- the .source object of the first input in the object .peak;
- the first input in the object .left;
- the second input in the object .right;

```
class SpanOfPartitions:
  #The objects of the class are:
  #.peak (list);
  #.source (MorphismOfPartitions);
  #.target (MorphismOfPartitions).
  def __init__(self,left_morphism,right_morphism):
  """ the source code of this constructor can be found in cl_sop.py """
```

If the pair of MorphismOfPartitions do not fit the desired format, the constructor outputs and error message and exits the program.

```
>>> a = MorphismOfPartitions([0,1,2,2,4,4,6],[0,1,3,3,4,4,4])
>>> b = MorphismOfPartitions([0,1,2,2,4,4,6],[2,1,2,2,2,2,6])
>>> span = SpanOfPartitions(a,b)
>>> print(span.peak)
[0, 1, 2, 2, 3, 3, 4]
>>> print(span.left.source)
[0, 1, 2, 2, 3, 3, 4]
>>> print(span.left.target)
[0, 1, 2, 2, 3, 3, 3]
>>> print(span.left.arrow)
[0, 1, 2, 3, 3]
>>> print(span.right.source)
[0, 1, 2, 2, 3, 3, 4]
>>> print(span.right.target)
0, 1, 0, 0, 0, 0, 2]
>>> print(span.right.arrow)
[0, 1, 0, 0, 2]
```

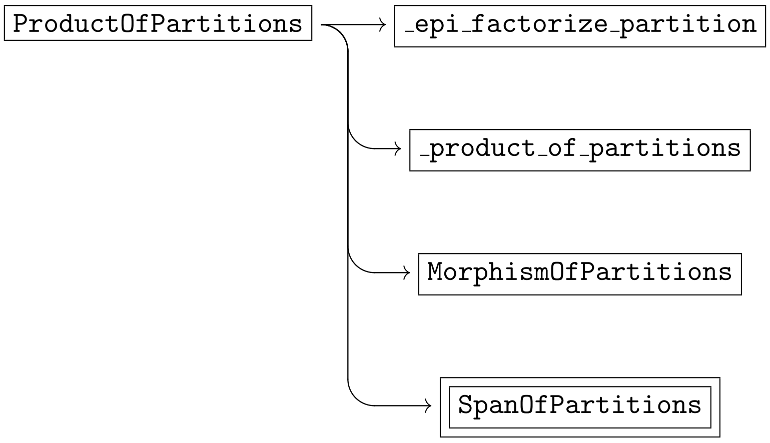
```
>>> b = MorphismOfPartitions([0,1,2,3,4,4,6],[2,1,2,2,2,2,6])
>>> span = SpanOfPartitions(a,b)
Error:  in SpanOfPartitions.__init__:  source objects do no match.
```

Note that **no new space** is allocated when storing the data of the two inputs, which means that if the one of the input lists is modified afterward, so is the span structure, as shown below.

```
>>> a.target[0] = 7
>>> print(span.left.target)
[7, 1, 2, 2, 3, 3, 3]
>>> a.source[0] = 21
>>> print(span.peak)
[21, 1, 2, 2, 3, 3, 4]
>>> b.source[0] = 8
>>> print(span.peak)
[21, 1, 2, 2, 3, 3, 4]
```

It is therefore important to be careful with the handling of list pointers. In partice, the constructor SpanOfPartitions.__init__ should be fed local (and temporary) variables so that the data of the span strucutre is not affected by external actions on the memory.

3.12. Description of ProductOfPartitions (class)



The class ProductOfPartitions possesses one object, namely .span (SpanOfPartitions) and a constructor .__init__. The consructor .__init__ takes two lists and stores the span structure of their categorical product (as partitions) in the object .span.

```
class ProductOfPartitions:
 #The object of the class is:
 #.span (SpanOfPartitions).
 def __init__(self,left_object,right_object):
 """ the source code of this constructor can be found in cl_pop.py """
```

If the lengths of the two input lists are not equal, the constructor outputs and error message and exits the program.

```
>>> p = ProductOfPartitions([0,1,3,3,4,4,4],[2,1,2,2,2,2,6])
>>> print(p.span.peak)
[0, 1, 2, 2, 3, 3, 4]
>>> print(p.span.left.source)
[0, 1, 2, 2, 3, 3, 4]
```

```
>>> print(p.span.left.target)
[0, 1, 2, 2, 3, 3, 3]
>>> print(p.span.left.arrow)
[0, 1, 2, 3, 3]
>>> print(p.span.right.source)
[0, 1, 2, 2, 3, 3, 4]
>>> print(p.span.right.target)
0, 1, 0, 0, 0, 0, 2]
>>> print(p.span.right.arrow)
[0, 1, 0, 0, 2]
>>> p = ProductOfPartitions([0,1,3,3,4,4,4],[2,1,2,2,2,2,6,2])
Error:  in ProductOfPartitions.__init__:  lengths do not match.
```

Chapter 4

Presentation of the module `PedigradCategory.py`

4.1. 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.

```python
def _read_alignment_file(name_of_file,reading_mode):
  """ the source code of this function can be found in raf.py """
  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

```
>Name of taxon1
This is a text

>Name of taxon2
This is a
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’]
```

4.2. Description of _is_column_trivial

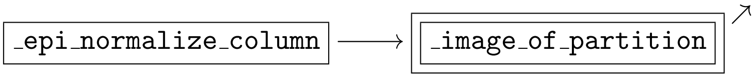
The function _is_column_trivial takes two lists of elements and returns 1 if, apart from the elements in the second input list, the first input list contains copies of a unique element; returns 0 if, apart from the elements in the second input list, the first input list contains at least two different elements.

```
def _is_column_trivial(column,exceptions):
  """ the source code of this function can be found in ict.py """
  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(_is_column_trivial(p))
1
>>> p = [‘e’,‘e’,‘a’,‘a’,‘e’]
>>> print(_is_column_trivial(p))
0
>>> p = [0,‘.’,0,0,‘.’,0,1,0]
>>> print(_is_column_trivial(p,[‘.’]))
0
>>> print(_is_column_trivial(p,[‘.’,1]))
1
```

4.3. Description of _epi_normalize_column



The function _epi_normalize_column refines the procedure

_epi_factorize_partition

of section 3.2 by relabeling the elements of a list with canonical labels except for those elements that can be found in a second input list. In other words, the procedure takes two input lists and returns one.

```
def _epi_normalize_column(column,exceptions):
  """ the source code of this function can be found in enc.py """
  return epimorphism
```

The first input lists is supposed to encode the column of an alignment file (which is usually stored in the object .local of a pedigrad; see section 4.4) while the second input list is the list of elements that should never be relabeled by the procedure (i.e. left as is).

```
>>> print(_epi_normalize_column([‘A’,‘4’,‘C’,‘C’,‘a’,‘A’],[‘4’]))
[0, ‘4’, 2, 2, 3, 0]
>>> print(_epi_normalize_column([‘A’,‘4’,‘C’,‘C’,‘a’,‘A’],[]))
[0, 1, 2, 2, 3, 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.

For illustration it is common that alignments contain ‘missing characters’ represented by the character ‘.’ as shown below.

Align.fa

```
>Alice
AGCTGACTG...AT
>Bob
ACGT..ACGATGAG
>Charles
TTCGA..AATCGCT
```

These missing characters are usually handled differently from the other characters, which is why one usually wants to avoid relabelling them – contrary to what the procedure of section 3.2 does.

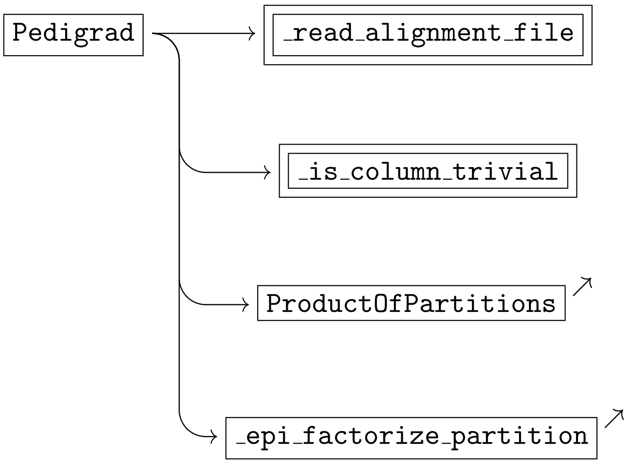
```
>>> P = Pedigrad("Align.fa",READ_DNA)
>>> for i in range(len(P.local)):
...    print(str(i)+":  "+str(P.local[i]))
0:  [0, 0, 1]
1:  [0, 1, 2]
2:  [0, 1, 0]
3:  [0, 0, 1]
4:  [0, ‘.’, 2]
5:  [0, ‘.’, ‘.’]
6:  [0, 1, ‘.’]
7:  [0, 1, 2]
8:  [0, 0, 1]
9:  [‘.’, 1, 2]
10:  [‘.’, 1, 2]
11:  [‘.’, 1, 1]
12:  [0, 0, 1]
13:  [0, 1, 0]
```

Obviously, the action of the program

$$\texttt{\_epi\_normalize\_column(}-\texttt{,[])}$$

on a list is the same as the action of the program _epi_factorize_partitions(−) on that same list. However, one of the reasons to use the latter would be to make the code more concise and clearer when only dealing with partitions. In addition, both previous procedures independently belong to their own module, which can turn out to be useful for migration and development purposes.

4.4. Description of Pedigrad (class)



The class Pedigrad possesses three objects, namely

- .local (list)
- .loci (list)
- .taxa (list)

and two methods, namely

- .__init__ (constructor)
- .partition

The consructor .__init__ takes the name of a file and an integer (as in the case of the function _read_alignment_file; see section 4.1) and stores

- the first output of _read_alignment_file in the object .taxa;
- the transpose of the second output of _read_alignment_file (when seen as a matrix), up to removal of those rows that contain the same character, in .local;
- the indices of the lists contained in .local as indexed the in the second output of _read_alignment_file in the object .loci (see below).

```
class Pedigrad:
  #The objects of the class are:
  #.local (list);
  #.loci (list);
  #.taxa (list).
  def __init__(self,name_of_file,reading_mode):
  """ the source code of this constructor can be found in cl_ped.py """
  def partition(self,segment):
  """ the source code of this constructor can be found in cl_ped.py """
    return the_image
```

If the empty string is given to the contructor .__init__, then an empty pedigrad is returned.

The method .partition takes a list of indices (or positions) within the range of the list stored in the object .local and returns the product of the partitions that are indexed by these positions in the object .local (i.e. its list).

If the input list fed to `.partition(−)` contains positions that are greater than or equal to the length of the list stored in `.local`, then the method outputs a error message and exits the program.

For illustration, let us consider the following alignment of characters `0`, `1` and `2` for the set of taxa: `Alice`, `Bob` and `Charles`. Note that the sequences associated with each taxa have the same legnth (i.e. 13 characters). The red characters show a column of the alignment whose characters are all the same.

`Align.fa`

```
>Alice
022...01002020
>Bob
00010102010222
>Charles
10100020101022
```

We can recover the list of taxa via the object `.taxa` and their associated sequences via the matrix contained in `.local` as shown below.

```
>>> P = Pedigrad("Align.fa",not(READ_DNA))
>>> for i in range(len(P.taxa)):
...    print(str(i)+":  "+str(P.taxa[i]))
0:  Alice
1:  Bob
2:  Charles
>>> for i in range(len(P.local)):
...    print(str(i)+":  "+str(P.local[i]))
0:  [‘0’, ‘0’, ‘1’]
1:  [‘2’, ‘0’, ‘0’]
2:  [‘2’, ‘0’, ‘1’]
3:  [‘.’, ‘1’, ‘0’]
4:  [‘.’, ‘0’, ‘0’]
5:  [‘.’, ‘1’, ‘0’]
6:  [‘0’, ‘0’, ‘2’]
7:  [‘1’, ‘2’, ‘0’]
8:  [‘0’, ‘0’, ‘1’]
9:  [‘0’, ‘1’, ‘0’]
10:  [‘2’, ‘0’, ‘1’]
11:  [‘0’, ‘2’, ‘0’]
12:  [‘0’, ‘2’, ‘2’]
```

Note that the list stored in `.taxa` do not contain the column that contained the same characters (see below). This information is memorized in the list contained by the object `.loci`, which does not contain the 12-th locus of the original alignment.

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

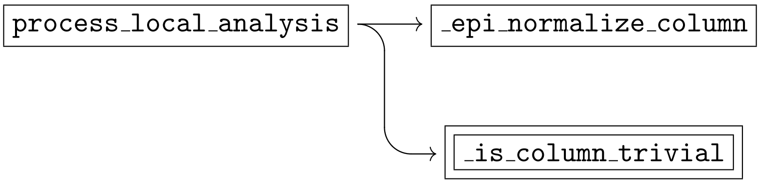
Note that position contained in `.loci` at a certain index i corresponds to the locus of the i-th list contained in `.local`.

The pedigrad structure then relates the three taxa `0` = `Alice`, `1` = `Bob`, `2` = `Charles` via various partitions. Whilen the object `.partition` only gives the images of the pedigrad, the groups relating the taxa can be displayed with the procedure `print_partition(−)`

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

```
>>> print_partition(P.partition([0]))
[[0, 1], [2]]
>>> print_partition(P.partition([1]))
[[0], [1, 2]]
>>> print_partition(P.partition([1,0]))
[[0], [1], [2]]
>>> print_partition(P.partition([1,0,13]))
Error: in Pedigrad.partition: index is not valid
```

4.5. Description of process_local_analysis



The function process_local_analysis takes three inputs:

- a list of lists meant to be the list stored in the object .local of a Pedigrad object;
- a list meant to be a list of special characters to be ignored during the procedure;
- a list of positions within the range of the first input list, and returns a lists of 2-tuples of lists.

```
def process_local_analysis(local_analysis,exceptions,positions):
  """ the source code of this function can be found in pla.py """
  return [(column,record)]+ process_local_analysis(
local_analysis,exceptions,new_positions)
```

Each 2-tuple of lists, say (l,m), contained in the output of process_local_analysis gives the list m of positions, that belong to the third input lists, at which the partition l can be found in the first input list, up to canonical relabeling.

The output process_local_analysis does not includes the columns that are trivial with respect to the special characters contained in the second input list (see section 4.2). Also, if the first input list is empty, then the procedure returns an empty list.

```
>>> l = [[0,1,0], [1,0,1], [3,3,4]]
>>> print(process_local_analysis(l,[],[0,1,2]))
[([0, 1, 0], [0, 1]), ([0, 0, 1], [2])]
>>> print(process_local_analysis(l,[‘.’],[0,2]))
[([0, 1, 0], [0]), ([0, 0, 1], [2])]
>>> l = [[0,1,0], [1,0,1], [3,3,‘.’]]
>>> print(process_local_analysis(l,[‘.’],[0,1,2]))
[([0, 1, 0], [0, 1])]
>>> l = [[0,1,0], [1,0,1], [3,3,3]]
>>> print(process_local_analysis(l,[‘.’],[0,1,2]))
[([0, 1, 0], [0, 1])]
```

In practice, the procedure process_local_analysis is meant to be used with the object .loci of a Pedigrad object. For illustration, consider the following alignment file, in which the red characters form trivial columns.

```
Align.fa
>Alice
YYNYN....UYUNMYUNMYUUYYY
>Bob
YUUUYMNMMYNYNM.....YMUNY
>Charles
UUYMNM.UNMYU....YMNUYUMN
```

First, we can use the Pedigrad class to analyze the previous alignment. As can be seen below, the indices of the trivial columns do not appear.

```
>>> P = Pedigrad("Align.fa",not(READ_DNA))
>>> proc = process_local_analysis(P.local,[‘.’],range(len(P.local)))
>>> for i in range(len(proc)):
...    print("type "+str(i)+":  "+str(proc[i]))
type 0:  ([0, 0, 1], [0, 23])
type 1:  ([0, 1, 1], [1, 21])
type 2:  ([0, 1, 2], [2, 3, 9, 20, 22])
type 3:  ([0, 1, 0], [4, 10, 11, 19])
type 4:  ([‘.’, 1, 2], [7, 8])
type 5:  ([0, ‘.’, 2], [16, 18])
```

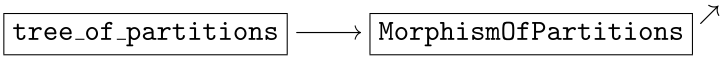
We can then use the list of indices P.loci with the indices given by the processed local analysis proc to recover the corresponding columns of the original alignment, as shown below for the columns of the form [0, 1, 2] (associated with type 2 given above).

```
>>> for i in proc[2][1]:
...    print("at locus "+str(P.loci[i])+":  "+str(P.local[P.loci[i]]))
at locus 2:  [‘N’, ‘U’, ‘Y’]
at locus 3:  [‘Y’, ‘U’, ‘M’]
at locus 9:  [‘U’, ‘Y’, ‘M’]
at locus 20:  [‘U’, ‘M’, ‘Y’]
at locus 22:  [‘Y’, ‘N’, ‘M’]
```

Chapter 5

Presentation of the module `AsciiTree.py`

5.1. 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.

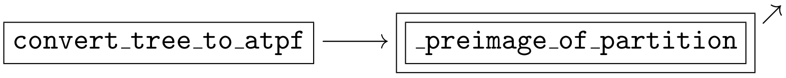
```python
def tree_of_partitions(partitions):
  """ the source code of this function can be found in top.py """
  return the_tree
```

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]
```

5.2. Description of convert_tree_to_atpf

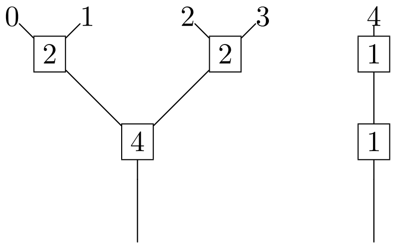


The function convert_tree_to_atpf takes a list of morphisms of partitions (as returned by the procedure tree_of_partitions) and converts it into its associated ascii tree preformat (abbrev. atpf), which is defined as any ATPF term that be can constructed from the following double grammar rules.

The grammar terms	and their associated weights
$\texttt{ATPF} := [\texttt{Tree}_1, \texttt{Tree}_2, \dots, \texttt{Tree}_k];$	
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Tree}_1, \texttt{Tree}_2, \dots, \texttt{Tree}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Tree}_i);$
$\texttt{Tree} := \big(\texttt{weight}(\texttt{Tree}), [\texttt{Leaf}_1, \texttt{Leaf}_2, \dots, \texttt{Leaf}_k]\big),$	$\texttt{weight}(\texttt{Tree}) := \sum_i \texttt{weight}(\texttt{Leaf}_i);$
$\texttt{Leaf} := (\texttt{weight}(\texttt{Leaf}), l),$ 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.

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

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`.

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

Since the depth is only returned for parsing purposes (see section 5.3 and section 5.4), 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 5.1).

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` → `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])])])]
```

5.3. Description of convert_atpf_to_atf

The function `convert_atpf_to_atf` takes an atpf and its depth (see section 5.2) 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.

```
def convert_atpf_to_atf(atpf,depth):
  """ the source code of this function can be found in cata.py """
  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 5.4) 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
```

5.4. Description of print_atf

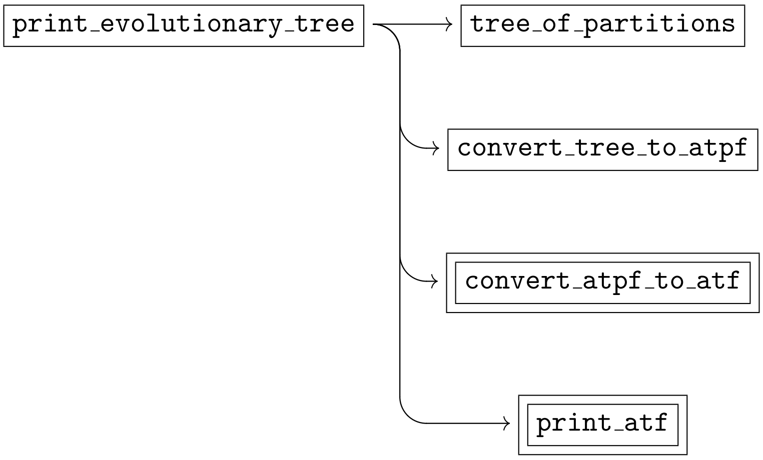
The function print_atf takes an atf and its depth and prints the ascii tree associated with the atf on the standard output.

```
def print_atf(atf,depth):
    """ 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[0],atpf[1])
>>> print_atf(atf,atpf[1])
|                |
|_____________   |
|             |  |
|______       |  |
|  |  |       |  |
A  C  D...E   F  B
```

5.5. 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.

```
def print_evolutionary_tree(atf,depth):
  #Returns a sequence of morphisms of partitions.
  tree = tree_of_partitions(partitions)
  #Returns an ascii tree pre-format and its depth.
  atpf = convert_tree_to_atpf(tree)
  #Returns the ascii tree format of the atpf.
  atf = convert_atpf_to_atf(atpf[0],atpf[1])
  #Prints the atf on the standard output.
  print_atf(atf,atpf[1])
```

Bibliography

[1] Rémy Tuyéras, *Categorical approach to tree inference*, arXiv:????

[2] —, *Category Theory for Genetics*, arXiv:1708.05255