Gfapy Documentation

Release 1.2.1

Giorgio Gonnella

Nov 10, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

The Graphical Fragment Assembly (GFA) are formats for the representation of sequence graphs, including assembly, variation and splicing graphs. Two versions of GFA have been defined (GFA1 and GFA2) and several sequence analysis programs have been adopting the formats as an interchange format, which allow to easily combine different sequence analysis tools.

This library implements the GFA1 and GFA2 specification described at https://github.com/GFA-spec/GFA-spec/blob/master/GFA-spec.md. It allows to create a Gfa object from a file in the GFA format or from scratch, to enumerate the graph elements (segments, links, containments, paths and header lines), to traverse the graph (by traversing all links outgoing from or incoming to a segment), to search for elements (e.g. which links connect two segments) and to manipulate the graph (e.g. to eliminate a link or a segment or to duplicate a segment distributing the read counts evenly on the copies).

The GFA format can be easily extended by users by defining own custom tags and record types. In Gfapy, it is easy to write extensions modules, which allow to define custom record types and datatypes for the parsing and validation of custom fields. The custom lines can be connected, using references, to each other and to lines of the standard record types.

1.1 Requirements

Gfapy has been written for Python 3 and tested using Python version 3.7. It does not require any additional Python packages or other software.

1.2 Installation

Gfapy is distributed as a Python package and can be installed using the Python package manager pip, as well as conda (in the Bioconda channel). It is also available as a package in some Linux distributions (Debian, Ubuntu).

The following command installs the current stable version from the Python Packages index:

```
pip install gfapy
```

If you would like to install the current development version from Github, use the following command:

```
pip install -e git+https://github.com/ggonnella/gfapy.git#egg=gfapy
```

Alternatively it is possible to install gfapy using conda. Gfapy is included in the Bioconda (https://bioconda.github.io/) channel:

```
conda install -c bioconda gfapy
```

1.3 Usage

If you installed gfapy as described above, you can import it in your script using the conventional Python syntax:

```
>>> import gfapy
```

1.4 Documentation

The documentation, including this introduction to Gfapy, a user manual and the API documentation is hosted on the ReadTheDocs server, at the URL http://gfapy.readthedocs.io/en/latest/ and it can be downloaded as PDF from the URL https://github.com/ggonnella/gfapy/blob/master/manual/gfapy-manual.pdf.

1.5 References

Giorgio Gonnella and Stefan Kurtz "GfaPy: a flexible and extensible software library for handling sequence graphs in Python", Bioinformatics (2017) btx398 https://doi.org/10.1093/bioinformatics/btx398

CHAPTER

TWO

CHANGELOG

```
== 1.2.1 ==

- fixed an issue with linear path merging (issue 21)
- GFA1 paths can contain a single segment only (issue 22)

== 1.2.0 ==

- fixed all open issues

== 1.1.0 ==

- fix: custom tags are not necessarily lower case
- additional support for rGFA subset of GFA1 by setting option dialect="rgfa"

== 1.0.0 ==

- initial release
```

CHAPTER

THREE

THE GFA CLASS

The content of a GFA file is represented in Gfapy by an instance of the class `Gfa`. In most cases, the Gfa instance will be constructed from the data contained in a GFA file, using the method `Gfa.from_file()`.

Alternatively, it is possible to use the construct of the class; it takes an optional positional parameter, the content of a GFA file (as string, or as list of strings, one per line of the GFA file). If no GFA content is provided, the Gfa instance will be empty.

```
>>> gfa = gfapy.Gfa("H\tVN:Z:1.0\nS\tA\t*")
>>> print(len(gfa.lines))
2
>>> gfa = gfapy.Gfa(["H\tVN:Z:1.0", "S\tA\t*", "S\tB\t*"])
>>> print(len(gfa.lines))
3
>>> gfa = gfapy.Gfa()
>>> print(len(gfa.lines))
0
```

The string representation of the Gfa object (which can be obtained using `str()`) is the textual representation in GFA format. Using `Gfa.to_file(filename)` allows writing this representation to a GFA file (the content of the file is overwritten).

```
>>> g1 = gfapy.Gfa()
>>> g1.append("H\tVN:Z:1.0")
>>> g1.append("S\ta\t*")
>>> g1.to_file("my.gfa")
>>> g2 = gfapy.Gfa.from_file("my.gfa")
>>> str(g1)
'H\tVN:Z:1.0\nS\ta\t*'
```

All methods for creating a Gfa (constructor and from_file) accept a `vlevel` parameter, the validation level, and can assume the values 0, 1, 2 and 3. A higher value means more validations are performed. The *Validation* chapter explains the meaning of the different validation levels in detail. The default value is 1.

```
>>> gfapy.Gfa().vlevel
1
>>> gfapy.Gfa(vlevel = 0).vlevel
0
```

A further parameter is `version`. It can be set to `'gfa1'`, `'gfa2'` or left to the default value (`None`). The default is to auto-detect the version of the GFA from the line content. If the version is set manually, any content not compatible to the specified version will trigger an exception. If the version is set automatically, an exception will be raised if two lines are found, with content incompatible to each other (e.g. a GFA1 segment followed by a GFA2 segment).

```
>>> g = gfapy.Gfa(version='gfa2')
>>> g.version
'gfa2'
>>> g.add_line("S\t1\t*")
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
gfapy.error.VersionError: Version: 1.0 (None)
...
>>> g = gfapy.Gfa()
>>> g.version
>>> g.add_line("S\t1\t*")
>>> g.version
'gfa1'
>>> g.add_line("S\t1\t100\t*")
Traceback (most recent call last):
...
gfapy.error.VersionError: Version: 1.0 (None)
...
```

3.1 Collections of lines

The property `lines` of the Gfa object is a list of all the lines in the GFA file (including the header, which is split into single-tag lines). The list itself shall not be modified by the user directly (i.e. adding and removing lines is done using a different interface, see below). However the single elements of the list can be edited.

```
>>> for line in gfa.lines: print(line)
```

For most record types, a list of the lines of the record type is available as a read-only property, which is named after the record type, in plural.

```
>>> [str(line) for line in gfa1.segments]
['S\t1\t*', 'S\t2\t*', 'S\t3\t*']
>>> [str(line) for line in gfa2.fragments]
[]
```

A particular case are edges; these are in GFA1 links and containments, while in GFA2 there is a unified edge record type, which also allows to represent internal alignments. In Gfapy, the `edges` property retrieves all edges (i.e. all E lines in GFA2, and all L and C lines in GFA1). The `dovetails` property is a list of all edges which represent dovetail overlaps (i.e. all L lines in GFA1 and a subset of the E lines in GFA2). The `containments` property is a list of all edges which represent containments (i.e. all C lines in GFA1 and a subset of the E lines in GFA2).

```
>>> gfa2.edges
[]
>>> gfa2.dovetails
[]
>>> gfa2.containments
[]
```

Paths are retrieved using the `paths` property. This list contains all P lines in GFA1 and all O lines in GFA2. Sets returns the list of all U lines in GFA2 (empty list in GFA1).

```
>>> gfa2.paths
[]
>>> gfa2.sets
[]
```

The header contain metadata in a single or multiple lines. For ease of access to the header information, all its tags are summarized in a single line instance, which is retrieved using the `header` property. This list The *The Header* chapter of this manual explains more in detail, how to work with the header object.

```
>>> gfa2.header.TS
100
```

All lines which start by the string # are comments; they are handled in the *Comments* chapter and are retrieved using the `comments` property.

```
>>> [str(line) for line in gfa1.comments]
['# this is a comment']
```

Custom lines are lines of GFA2 files which start with a non-standard record type. Gfapy provides basic built-in support for accessing the information in custom lines, and allows to define extensions for own record types for defining more advanced functionality (see the *Custom records* chapter).

```
>>> [str(line) for line in gfa2.custom_records]
['X\tcustom line', 'Y\tcustom line']
>>> gfa2.custom_record_keys
['X', 'Y']
>>> [str(line) for line in gfa2.custom_records_of_type('X')]
['X\tcustom line']
```

3.2 Line identifiers

Some GFA lines have a mandatory or optional identifier field: segments and paths in GFA1, segments, gaps, edges, paths and sets in GFA2. A line of this type can be retrieved by identifier, using the method `Gfa.line(ID)` using the identifier as argument.

```
>>> str(gfa1.line('1'))
'S\t1\t*'
```

The GFA2 specification prescribes the exact namespace for the identifier (segments, paths, sets, edges and gaps identifier share the same namespace). The content of this namespace can be retrieved using the `names` property. The identifiers of single line types can be retrieved using the properties `segment_names`, `edge_names`, `gap_names`, `path_names` and `set_names`.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t100\t*")
>>> g.add_line("S\tB\t100\t*")
>>> g.add_line("S\tC\t100\t*")
>>> g.add_line("E\tb_c\tB+\tC+\t0\t10\t90\t100$\t*")
>>> g.add_line("O\tp1\tB+ C+")
>>> g.add_line("U\ts1\tA b_c g")
>>> g.add_line("G\tg\tA+\tB-\t1000\t*")
>>> g.names
['A', 'B', 'C', 'b_c', 'g', 'p1', 's1']
>>> g.segment_names
['A', 'B', 'C']
>>> g.path_names
['p1']
>>> g.edge_names
['b_c']
>>> g.gap_names
['g']
>>> g.set_names
['s1']
```

The GFA1 specification does not handle the question of the namespace of identifiers explicitly. However, gfapy assumes and enforces a single namespace for segment, path names and the values of the ID tags of L and C lines. The content of this namespace can be found using `names` property. The identifiers of single line types

can be retrieved using the properties `segment_names`, `edge_names` (ID tags of links and containments) and `path_names`. For GFA1, the properties `gap_names`, `set_names` contain always empty lists.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t*")
>>> g.add_line("S\tB\t*")
>>> g.add_line("S\tC\t*")
>>> g.add_line("L\tB\t+\tC\t+\t*\tID:Z:b_c")
>>> g.add_line("P\tp1\tB+,C+\t*")
>>> g.names
['A', 'B', 'C', 'b_c', 'p1']
>>> g.segment_names
['A', 'B', 'C']
>>> g.path_names
['p1']
>>> g.edge_names
['b_c']
>>> g.gap_names
[]
>>> g.set_names
[]
```

3.3 Identifiers of external sequences

Fragments contain identifiers which refer to external sequences (not contained in the GFA file). According to the specification, the these identifiers are not part of the same namespace as the identifier of the GFA lines. They can be retrieved using the `external_names` property.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t100\t*")
>>> g.add_line("F\tA\tread1+\t10\t30\t0\t20$\t20M")
>>> g.external_names
['read1']
```

The method `Gfa.fragments_for_external(external_ID)` retrieves all F lines with a specified external sequence identifier.

```
>>> f = g.fragments_for_external('read1')
>>> len(f)
1
>>> str(f[0])
'F\tA\tread1+\t10\t30\t0\t20$\t20M'
```

3.4 Adding new lines

New lines can be added to a Gfa instance using the `Gfa.add_line(line)` method or its alias `Gfa.append(line)`. The argument can be either a string describing a line with valid GFA syntax, or a `Line` instance. If a string is added, a line instance is created and then added.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t*")
>>> g.segment_names
['A']
>>> g.append("S\tB\t*")
>>> g.segment_names
['A', 'B']
```

3.5 Editing the lines

Accessing the information stored in the fields of a line instance is described in the *Positional fields* and *Tags* chapters.

In Gfapy, a line instance belonging to a Gfa instance is said to be *connected* to the Gfa instance. Direct editing the content of a connected line is only possible, for those fields which do not contain references to other lines. For more information on how to modify the content of the fields of connected line, see the *References* chapter.

```
>>> g = gfapy.Gfa()
>>> e = gfapy.Line("E\t*\tA+\tB-\t0\t10\t90\t100$\t*")
>>> e.sid1 = "C+"
>>> g.add_line(e)
>>> e.sid1 = "A+"
Traceback (most recent call last):
gfapy.error.RuntimeError: ...
```

3.6 Removing lines

Disconnecting a line from the Gfa instance is done using the `Gfa.rm(line)` method. The argument can be a line instance or the name of a line.

In alternative, a line instance can also be disconnected using the `disconnect` method on it. Disconnecting a line may trigger other operations, such as the disconnection of other lines (see the *References* chapter).

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t*")
>>> g.segment_names
['A']
>>> g.rm('A')
>>> g.segment_names
[]
>>> g.append("S\tB\t*")
>>> g.segment_names
['B']
>>> b = g.line('B')
>>> b.disconnect()
>>> g.segment_names
[]
```

3.7 Renaming lines

Lines with an identifier can be renamed. This is done simply by editing the corresponding field (such as `name` or `sid` for a segment). This field is not a reference to another line and can be freely edited also in line instances connected to a Gfa. All references to the line from other lines will still be up to date, as they will refer to the same instance (whose name has been changed) and their string representation will use the new name.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\tA\t*")
>>> g.add_line("L\tA\t+\tB\t-\t*")
>>> g.segment_names
['A', 'B']
>>> g.dovetails[0].from_name
'A'
>>> g.segment('A').name = 'C'
>>> g.segment_names
```

(continues on next page)

(continued from previous page)

```
['B', 'C']
>>> g.dovetails[0].from_name
'C'
```

CHAPTER

FOUR

VALIDATION

Different validation levels are available. They represent different compromises between speed and warrant of validity. The validation level can be specified when the `Gfa` object is created, using the `vlevel` parameter of the constructor and of the `from_file` method. Four levels of validation are defined (0 = no validation, 1 = validation by reading, 2 = validation by reading and writing, 3 = continuous validation). The default validation level value is 1.

4.1 Manual validation

Independently from the validation level chosen, the user can always check the value of a field calling `validate_field()` on the line instance. If no exception is raised, the field content is valid.

To check if the entire content of the line is valid, the user can call `validate()` on the line instance. This will check all fields and perform cross-field validations, such as comparing the length of the sequence of a GFA1 segment, to the value of the LN tag (if present).

It is also possible to validate the structure of the GFA, for example to check if there are unresolved references to lines. To do this, use the `validate()` of the `Gfa` instance.

4.2 No validations

If the validation is set to 0, Gfapy will try to accept any input and never raise an exception. This is not always possible, and in some cases, an exception will still be raised, if the data is invalid.

4.3 Validation when reading

If the validation level is set to 1 or higher, basic validations will be performed, such as checking the number of positional fields, the presence of duplicated tags, the tag datatype of predefined tags. Additionally, all tags will be validated, either during parsing or on first access. Record-type cross-field validations will also be performed.

In other words, a validation of 1 means that Gfapy guarantees (as good as it can) that the GFA content read from a file is valid, and will raise an exception on accessing the data if not.

The user is supposed to call `validate_field` after changing a field content to something which can be potentially invalid, or `validate()` if potentially cross-field validations could fail.

4.4 Validation when writing

Setting the level to 2 will perform all validations described above, plus validate the fields content when their value is written to string.

In other words, a validation of 2 means that Gfapy guarantee (as good as it can) that the GFA content read from a file and written to a file is valid and will raise an exception on accessing the data or writing to file if not.

4.5 Continuous validation

If the validation level is set to 3, all validations for lower levels described above are run, plus a validation of fields contents each time a setter method is used.

A validation of 3 means that Gfapy guarantees (as good as it can) that the GFA content is always valid.

CHAPTER

FIVE

POSITIONAL FIELDS

Most lines in GFA have positional fields (Headers are an exception). During parsing, if a line is encountered, which has too less or too many positional fields, an exception will be thrown. The correct number of positional fields is record type-specific.

Positional fields are recognized by its position in the line. Each positional field has an implicit field name and datatype associated with it.

5.1 Field names

The field names are derived from the specification. Lower case versions of the field names are used and spaces are substituted with underscores. In some cases, the field names were changed, as they represent keywords in common programming languages (`from`, `send`).

The following tables shows the field names used in Gfapy, for each kind of line. Headers have no positional fields. Comments and custom records follow particular rules, see the respective chapters (*Comments* and *Custom records*).

5.1.1 GFA1 field names

Record Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
Segment	`name`	`sequence`				
Link	`from_segment`	`from_orient`	`to_segment`	`to_orient`	`overlap`	
Containment	`from_segment`	`from_orient`	`to_segment`	`to_orient`	`pos`	`overlap`
Path	`path_name`	`segment_names`	`overlaps`			

5.1.2 GFA2 field names

Record Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6	Field 7	Field 8
Segment	`sid`	`slen`	`sequence`					
Edge	`eid`	`sid1`	`sid2`	`beg1`	`end1`	`beg2`	`end2`	`alignment`
Fragment	`sid`	`external`	`s_beg`	`s_end`	`f_beg`	`f_end`	`alignment`	
Gap	`gid`	`sid1`	`d1`	`d2`	`sid2`	`disp`	`var`	
Set	`pid`	`items`						
Path	`pid`	`items`						

5.2 Datatypes

The datatype of each positional field is described in the specification and cannot be changed (differently from tags). Here is a short description of the Python classes used to represent data for different datatypes.

5.2.1 Placeholders

The positional fields in GFA can never be empty. However, there are some fields with optional values. If a value is not specified, a placeholder character is used instead (`*`). Such undefined values are represented in Gfapy by the `Placeholder` class, which is described more in detail in the *Placeholders* chapter.

5.2.2 Arrays

The `items` field in unordered and ordered groups and the `segment_names` and `overlaps` fields in paths are lists of objects and are represented by list instances.

```
>>> set = gfapy.Line("U\t*\t1 A 2")
>>> type(set.items)
<class 'list'>
>>> gfa2_path = gfapy.Line("O\t*\tA+ B-")
>>> type(gfa2_path.items)
<class 'list'>
>>> gfa1_path = gfapy.Line("P\tp1\tA+,B-\t10M,9M1D1M")
>>> type(gfa1_path.segment_names)
<class 'list'>
>>> type(gfa1_path.overlaps)
<class 'list'>
```

5.2.3 Orientations

Orientations are represented by strings. The `gfapy.invert()` method applied to an orientation string returns the other orientation.

```
>>> gfapy.invert("+")
'-'
>>> gfapy.invert("-")
'+'
```

5.2.4 Identifiers

The identifier of the line itself (available for S, P, E, G, U, O lines) can always be accessed in Gfapy using the `name` alias and is represented in Gfapy by a string. If it is optional (E, G, U, O lines) and not specified, it is represented by a Placeholder instance. The fragment identifier is also a string.

Identifiers which refer to other lines are also present in some line types (L, C, E, G, U, O, F). These are never placeholders and in stand-alone lines are represented by strings. In connected lines they are references to the Line instances to which they refer to (see the *References* chapter).

5.2.5 Oriented identifiers

Oriented identifiers (e.g. `segment_names` in GFA1 paths) are represented by elements of the class `gfapy.OrientedLine`. The `segment` method of the oriented segments returns the segment identifier (or segment reference in connected path lines) and the `orient` method returns the orientation string. The `name` method returns the string of the segment, even if this is a reference to a segment. A new oriented line can be created using the `OL[line, orientation]` method.

Calling `invert` returns an oriented segment, with inverted orientation. To set the two attributes the methods `segment=` and `orient=` are available.

Examples:

```
>>> p = gfapy.Line("P\tP1\ta+,b-\t*")
>>> p.segment_names
[gfapy.OrientedLine('a','+'), gfapy.OrientedLine('b','-')]
>>> sn0 = p.segment_names[0]
>>> sn0.line
'a'
>>> sn0.name
'a'
>>> sn0.orient
'+'
>>> sn0.invert()
>>> sn0
gfapy.OrientedLine('a','-')
>>> sn0.orient
'-'
>>> sn0.line = gfapy.Line('S\tX\t*')
>>> str(sn0)
'X-'
>>> sn0.name
'X'
>>> sn0 = gfapy.OrientedLine(gfapy.Line('S\tY\t*'), '+')
```

5.2.6 Sequences

Sequences (S field sequence) are represented by strings in Gfapy. Depending on the GFA version, the alphabet definition is more or less restrictive. The definitions are correctly applied by the validation methods.

The method `rc()` is provided to compute the reverse complement of a nucleotidic sequence. The extended IUPAC alphabet is understood by the method. Applied to non nucleotidic sequences, the results will be meaningless:

```
>>> from gfapy.sequence import rc
>>> rc("gcat")
'atgc'
>>> rc("*")
'*'
>>> rc("yatc")
'gatr'
>>> rc("gCat")
'atGc'
>>> rc("cag", rna=True)
'cug'
```

5.2.7 Integers and positions

The C lines `pos` field and the G lines `disp` and `var` fields are represented by integers. The `var` field is optional, and thus can be also a placeholder. Positions are 0-based coordinates.

The position fields of GFA2 E lines (`beg1, beg2, end1, end2`) and F lines (`s_beg, s_end, f_beg, f_end`) contain a dollar string as suffix if the position is equal to the segment length. For more information, see the *Positions* chapter.

5.2.8 Alignments

Alignments are always optional, ie they can be placeholders. If they are specified they are CIGAR alignments or, only in GFA2, trace alignments. For more details, see the *Alignments* chapter.

5.2.9 GFA1 datatypes

Datatype	Record Type	Fields
Identifier	Segment	`name`
	Path	`path_name`
	Link	`from_segment, to_segment`
	Containment	`from_segment, to_segment`
[OrientedIdentifier]	Path	`segment_names`
Orientation	Link	`from_orient, to_orient`
	Containment	`from_orient, to_orient`
Sequence	Segment	`sequence`
Alignment	Link	`overlap`
	Containment	`overlap`
[Alignment]	Path	`overlaps`
Position	Containment	`pos`

5.2.10 GFA2 datatypes

Datatype	Record Type	Fields
Itentifier	Segment	`sid`
	Fragment	`sid`
OrientedIdentifier	Edge	`sid1, sid2`
	Gap	`sid1, sid2`
	Fragment	`external`
OptionalIdentifier	Edge	`eid`
	Gap	`gid`
	U Group	`oid`
	O Group	`uid`
[Identifier]	U Group	`items`
[OrientedIdentifier]	O Group	`items`
Sequence	Segment	`sequence`
Alignment	Edge	`alignment`
	Fragment	`alignment`
Position	Edge	`beg1, end1, beg2, end2`
	Fragment	`s_beg, s_end, f_beg, f_end`
Integer	Gap	`disp, var`

5.3 Reading and writing positional fields

The `positional_fieldnames` method returns the list of the names (as strings) of the positional fields of a line. The positional fields can be read using a method on the Gfapy line object, which is called as the field name. Setting the value is done with an equal sign version of the field name method (e.g. segment.slen = 120). In alternative, the `set(fieldname, value)` and `get(fieldname)` methods can also be used.

```
>>> s_gfa1 = gfapy.Line("S\t1\t*")
>>> s_gfa1.positional_fieldnames
['name', 'sequence']
>>> s_gfa1.name
'1'
>>> s_gfa1.get("name")
'1'
>>> s_gfa1.name = "segment2"
>>> s_gfa1.name
'segment2'
>>> s_gfa1.set('name',"3")
>>> s_gfa1.name
'3'
```

When a field is read, the value is converted into an appropriate object. The string representation of a field can be read using the `field_to_s(fieldname)` method.

```
>>> gfa = gfapy.Gfa()
>>> gfa.add_line("S\ts1\t*")
>>> gfa.add_line("L\ts1\t+\ts2\t-\t*")
>>> link = gfa.dovetails[0]
>>> str(link.from_segment)
'S\ts1\t*'
>>> link.field_to_s('from_segment')
's1'
```

When setting a non-string field, the user can specify the value of a tag either as a Python non-string object, or as the string representation of the value.

```
>>> gfa = gfapy.Gfa(version='gfa1')
>>> gfa.add_line("C\ta\t+\tb\t-\t10\t*")
>>> c = gfa.containments[0]
>>> c.pos
10
>>> c.pos = 1
>>> c.pos
1
>>> c.pos = "2"
>>> c.pos
2
>>> c.field_to_s("pos")
'2'
```

Note that setting the value of reference and backreferences-related fields is generally not allowed, when a line instance is connected to a Gfa object (see the *References* chapter).

```
>>> gfa = gfapy.Gfa(version='gfa1')
>>> l = gfapy.Line("L\ts1\t+\ts2\t-\t*")
>>> l.from_name
's1'
>>> l.from_segment = "s3"
>>> l.from_name
's3'
>>> gfa.add_line(l)
```

(continues on next page)

(continued from previous page)

```
>>> l.from_segment = "s4"
Traceback (most recent call last):
...
gfapy.error.RuntimeError: ...
```

5.4 Validation

The content of all positional fields must be a correctly formatted string according to the rules given in the GFA specifications (or a Python object whose string representation is a correctly formatted string).

Depending on the validation level, more or less checks are done automatically (see the *Validation* chapter). Not regarding which validation level is selected, the user can trigger a manual validation using the `validate_field(fieldname)` method for a single field, or using `validate`, which does a full validation on the whole line, including all positional fields.

```
>>> line = gfapy.Line("H\txx:i:1")
>>> line.validate_field("xx")
>>> line.validate()
```

5.5 Aliases

For some fields, aliases are defined, which can be used in all contexts where the original field name is used (i.e. as parameter of a method, and the same setter and getter methods defined for the original field name are also defined for each alias, see below).

```
>>> gfa1_path = gfapy.Line("P\tX\t1-,2+,3+\t*")
>>> gfa1_path.name == gfa1_path.path_name
True
>>> edge = gfapy.Line("E\t*\tA+\tB-\t0\t10\t90\t100$\t*")
>>> edge.eid == edge.name
True
>>> containment = gfapy.Line("C\tA\t+\tB\t-\t10\t*")
>>> containment.from_segment == containment.container
True
>>> segment = gfapy.Line("S\t1\t*")
>>> segment.sid == segment.name
True
>>> segment.sid
'1'
>>> segment.name = '2'
>>> segment.sid
'2'
```

5.5.1 Name

Different record types have an identifier field: segments (name in GFA1, sid in GFA2), paths (path_name), edge (eid), fragment (sid), gap (gid), groups (pid).

All these fields are aliased to `name`. This allows the user for example to set the identifier of a line using the `name=(value)` method using the same syntax for different record types (segments, edges, paths, fragments, gaps and groups).

5.5.2 Version-specific field names

For segments the GFA1 name and the GFA2 sid are equivalent fields. For this reason an alias `sid` is defined for GFA1 segments and `name` for GFA2 segments.

5.5.3 Crypical field names

The definition of from and to for containments is somewhat cryptic. Therefore following aliases have been defined for containments: container[_orient] for from[_|segment|orient]; contained[_orient] for to[_segment|orient].

CHAPTER

SIX

PLACEHOLDERS

Some positional fields may contain an undefined value S: `sequence`; L/C: `overlap`; P: `overlaps`; E: `eid`, `alignment`; F: `alignment`; G: `gid`, `var`; U/O: `pid`. In GFA this value is represented by a `*`.

In Gfapy the class `Placeholder` represent the undefined value.

6.1 Distinguishing placeholders

The `gfapy.is_placeholder()` method allows to check if a value is a placeholder; a value is a placeholder if it is a `Placeholder` instance, or would represent a placeholder in GFA (a string containing `*`), or would be represented by a placeholder in GFA (e.g. an empty array).

```
>>> gfapy.is_placeholder("*")
True
>>> gfapy.is_placeholder("**")
False
>>> gfapy.is_placeholder([])
True
>>> gfapy.is_placeholder(gfapy.Placeholder())
True
```

Note that, as a placeholder is `False` in boolean context, just a `if not placeholder` will also work, if the value is an instance of `Placeholder`, but not always for the other cases (in particular not for the string representation `*`). Therefore using `gfapy.is_placeholder()` is better.

```
>>> if "*": print('* is not a placeholder')
* is not a placeholder
>>> if gfapy.is_placeholder("*"): print('but it represents a placeholder')
but it represents a placeholder
```

6.2 Compatibility methods

Some methods are defined for placeholders, which allow them to respond to the same methods as defined values. This allows to write generic code.

```
>>> placeholder = gfapy.Placeholder()
>>> placeholder.validate() # does nothing
>>> len(placeholder)
0
>>> placeholder[1]
gfapy.Placeholder()
>>> placeholder + 1
gfapy.Placeholder()
```

CHAPTER

SEVEN

POSITIONS

The only position field in GFA1 is the `pos` field in the C lines. This represents the starting position of the contained segment in the container segment and is 0-based.

Some fields in GFA2 E lines (`beg1`, `beg2`, `end1`, `end2`) and F lines (`s_beg`, `s_end`, `f_beg`, `f_end`) are positions. According to the specification, they are 0-based and represent virtual ticks before and after each string in the sequence. Thus ranges are represented similarly to the Python range conventions: e.g. a 1-character prefix of a sequence will have begin 0 and end 1.

7.1 Last positions in GFA2

The GFA2 positions must contain an additional string (`$`) appended to the integer, if (and only if) they are the last position in the segment sequence. These particular positions are represented in Gfapy as instances of the class `LastPos`.

To create a lastpos instance, the constructor can be used with an integer, or the string representation (which must end with the dollar sign, otherwise an integer is returned):

```
>>> str(gfapy.LastPos(12))
'12$'
>>> gfapy.LastPos("12")
12
>>> str(gfapy.LastPos("12"))
'12'
>>> gfapy.LastPos("12$")
gfapy.LastPos(12)
>>> str(gfapy.LastPos("12$"))
'12$'
```

Subtracting an integer from a lastpos returns a lastpos if 0 subtracted, an integer otherwise. This allows to do some arithmetic on positions without making them invalid.

```
>>> gfapy.LastPos(12) - 0
gfapy.LastPos(12)
>>> gfapy.LastPos(12) - 1
11
```

The functions `islastpos()` and `isfirstpos()` allow to determine if a position value is 0 (first), or the last position, using the same syntax for lastpos and integer instances.

```
>>> gfapy.isfirstpos(0)
True
>>> gfapy.islastpos(0)
False
>>> gfapy.isfirstpos(12)
False
>>> gfapy.islastpos(12)
```

(continues on next page)

(continued from previous page)

```
False
>>> gfapy.islastpos(gfapy.LastPos("12"))
False
>>> gfapy.islastpos(gfapy.LastPos("12$"))
True
```

CHAPTER

EIGHT

ALIGNMENTS

Some GFA1 (L/C overlap, P overlaps) and GFA2 (E/F alignment) fields contain alignments or lists of alignments. The alignment can be left unspecified and a placeholder symbol `*` used instead. In GFA1 the alignments can be given as CIGAR strings, in GFA2 also as Dazzler traces.

Gfapy uses three different classes for representing the content of alignment fields: `CIGAR`, `Trace` and `AlignmentPlaceholder`.

8.1 Creating an alignment

An alignment instance is usually created from its GFA string representation or from a list by using the `gfapy.Alignment()` constructor.

```
>>> from gfapy import Alignment
>>> Alignment("*")
gfapy.AlignmentPlaceholder()
>>> Alignment("10,10,10")
gfapy.Trace([10,10,10])
>>> Alignment([10,10,10])
gfapy.Trace([10,10,10])
>>> Alignment("30M2I")
gfapy.CIGAR([gfapy.CIGAR.Operation(30,'M'), gfapy.CIGAR.Operation(2,'I')])
```

If the argument is an alignment object it will be returned, so that is always safe to call the method on a variable which can contain a string or an alignment instance:

```
>>> Alignment(Alignment("*"))
gfapy.AlignmentPlaceholder()
>>> Alignment(Alignment("10,10"))
gfapy.Trace([10,10])
```

8.2 Recognizing undefined alignments

The `gfapy.is_placeholder()` method allows to test if an alignment field contains an undefined value (placeholder) instead of a defined value (CIGAR string, trace). The method accepts as argument either an alignment object or a string or list representation.

```
>>> from gfapy import is_placeholder, Alignment
>>> is_placeholder(Alignment("30M"))
False
>>> is_placeholder(Alignment("10,10"))
False
>>> is_placeholder(Alignment("*"))
True
```

(continues on next page)

(continued from previous page)

```
>>> is_placeholder("*")
True
>>> is_placeholder("30M")
False
>>> is_placeholder("10,10")
False
>>> is_placeholder([])
True
>>> is_placeholder([10,10])
False
```

Note that, as a placeholder is `False` in boolean context, just a `if not aligment` will also work, if alignment is an alignment object. But this of course, does not work, if it is a string representation. Therefore it is better to use the `gfapy.is_placeholder()` method, which works in both cases.

```
>>> if not Alignment("*"): print('no alignment')
no alignment
>>> if is_placeholder(Alignment("*")): print('no alignment')
no alignment
>>> if "*": print('not a placeholder...?')
not a placeholder...?
>>> if is_placeholder("*"): print('really? it is a placeholder!')
really? it is a placeholder!
```

8.3 Reading and editing CIGARs

CIGARs are represented by specialized lists, instances of the class `CIGAR`, whose elements are CIGAR operations CIGAR operations are represented by instance of the class `Operation`, and provide the properties `length` (length of the operation, an integer) and `code` (one-letter string which specifies the type of operation). Note that not all operations allowed in SAM files (for which CIGAR strings were first defined) are also meaningful in GFA and thus GFA2 only allows the operations `M`, `I`, `D` and `P`.

```
>>> cigar = gfapy.Alignment("30M")
>>> isinstance(cigar, list)
True
>>> operation = cigar[0]
>>> type(operation)
<class 'gfapy.alignment.cigar.CIGAR.Operation'>
>>> operation.code
'M'
>>> operation.code = 'D'
>>> operation.length
30
>>> len(operation)
30
>>> str(operation)
'30D'
```

As a CIGAR instance is a list, list methods apply to it. If the array is emptied, its string representation will be the placeholder symbol `*`.

```
>>> cigar = gfapy.Alignment("1I20M2D")
>>> cigar[0].code = "M"
>>> cigar.pop(1)
gfapy.CIGAR.Operation(20,'M')
>>> str(cigar)
'1M2D'
>>> cigar[:] = []
```

(continues on next page)

(continued from previous page)

```
>>> str(cigar)
'*'
```

The validate `CIGAR.validate()` function checks if a CIGAR instance is valid. A version can be provided, as the CIGAR validation is version specific (as GFA2 forbids some CIGAR operations).

```
>>> cigar = gfapy.Alignment("30M10D20M5I10M")
>>> cigar.validate()
>>> cigar[1].code = "L"
>>> cigar.validate()
Traceback (most recent call last):
  ...
gfapy.error.ValueError:
>>> cigar = gfapy.Alignment("30M10D20M5I10M")
>>> cigar[1].code = "X"
>>> cigar.validate(version="gfa1")
>>> cigar.validate(version="gfa2")
Traceback (most recent call last):
  ...
gfapy.error.ValueError:
```

8.4 Reading and editing traces

Traces are arrays of non-negative integers. The values are interpreted using a trace spacing value. If traces are used, a trace spacing value must be defined in a TS integer tag, either in the header, or in the single lines which contain traces (which takes precedence over the header global value).

```
>>> print(gfa)
H TS:i:100
E x A+ B- 0 100$ 0 100$ 4,2 TS:i:50
...
>>> gfa.header.TS
100
>>> gfa.line("x").TS
50
```

8.5 Query, reference and complement

CIGARs are asymmetric, i.e.they consider one sequence as reference and another sequence as query.

The `length_on_reference()` and `length_on_query()` methods compute the length of the alignment on the two sequences. These methods are used by the library e.g. to convert GFA1 L lines to GFA2 E lines (which is only possible if CIGARs are provided).

```
>>> cigar = gfapy.Alignment("30M10D20M5I10M")
>>> cigar.length_on_reference()
70
>>> cigar.length_on_query()
65
```

CIGARs are dependent on which sequence is taken as reference and which is taken as query. For each alignment, a complement CIGAR can be computed using the method `complement()`; it is the CIGAR obtained when the two sequences are switched.

```
>>> cigar = gfapy.Alignment("2M1D3M")
>>> str(cigar.complement())
'3M1I2M'
```

The current version of Gfapy does not provide a way to compute the alignment, thus the trace information can be accessed and edited, but not used for this purpose. Because of this there is currently no way in Gfapy to compute a complement trace (trace obtained when the sequences are switched).

```
>>> trace = gfapy.Alignment("1,2,3")
>>> str(trace.complement())
'*'
```

The complement of a placeholder is a placeholder:

```
>>> str(gfapy.Alignment("*").complement())
'*'
```

CHAPTER

NINE

TAGS

Each record in GFA can contain tags. Tags are fields which consist in a tag name, a datatype and data. The format is `NN:T:DATA` where `NN` is a two-letter tag name, `T` is a one-letter datatype string and `DATA` is a string representing the data according to the specified datatype. Tag names must be unique for each line, i.e. each line may only contain a tag once.

```
# Examples of GFA tags of different datatypes:
"aa:i:-12"
"bb:f:1.23"
"cc:Z:this is a string"
"dd:A:X"
"ee:B:c,12,3,2"
"ff:H:122FA0"
'gg:J:["A","B"]'
```

9.1 Custom tags

Some tags are explicitly defined in the specification (these are named *predefined tags* in Gfapy), and the user or an application can define its own custom tags. These may contain lower case letters.

Custom tags are user or program specific and may of course collide with the tags used by other users or programs. For this reasons, if you write scripts which employ custom tags, you should always check that the values are of the correct datatype and plausible.

```
>>> line = gfapy.Line("H\txx:i:2")
>>> if line.get_datatype("xx") != "i":
...   raise Exception("I expected the tag xx to contain an integer!")
>>> myvalue = line.xx
>>> if (myvalue > 120) or (myvalue % 2 == 1):
...   raise Exception("The value in the xx tag is not an even value <= 120")
>>> # ... do something with myvalue
```

Also it is good practice to allow the user of the script to change the name of the custom tags. For example, Gfapy employs the +or+ custom tag to track the original segment from which a segment in the final graph is derived. All methods which read or write the +or+ tag allow to specify an alternative tag name to use instead of +or+, for the case that this name collides with the custom tag of another program.

```
# E.g. a method which does something with myvalue, usually stored in tag xx
# allows the user to specify an alternative name for the tag
def mymethod(line, mytag="xx"):
  myvalue = line.get(mytag)
  # ...
```

9.2 Predefined tags

According to the GFA specifications, predefined tag names consist of either two upper case letters, or an upper case letter followed by a digit. The GFA1 specification predefines tags for each line type, while GFA2 only predefines tags for the header and edges.

While tags with the predefined names are allowed to be added to any line, when they are used in the lines mentiones in the specification (e.g. `VN` in the header) gfapy checks that the datatype is the one prescribed by the specification (e.g. `VN` must be of type `Z`). It is not forbidden to use the same tags in other contexts, but in this case, the datatype restriction is not enforced.

Tag | Type		Line types	GFA version
VN | Z		H	1,2
TS	i	H,S	2
LN	i	S	1
RC	i	S,L,C	1
FC	i	S,L	1
KC	i	S,L	1
SH	H	S	1
UR	Z	S	1
MQ	i	L	1
NM	i	L,i	1
ID	Z	L,C	1

```
"VN:Z:1.0" # VN => predefined tag
"z5:Z:1.0" # z5 first char is downcase => custom tag
"XX:Z:aaa" # XX upper case, but not predefined => custom tag

# not forbidden, but not recommended:
"zZ:Z:1.0" # => mixed case, first char downcase => custom tag
"Zz:Z:1.0" # => mixed case, first char upcase => custom tag
"vn:Z:1.0" # => same name as predefined tag, but downcase => custom tag
```

9.3 Datatypes

The following table summarizes the datatypes available for tags:

Symbol	Datatype	Example	Python class
Z	string	This is a string	str
i	integer	-12	int
f	float	1.2E-5	float
A	char	X	str
J	JSON	[1,{“k1”:1,”k2”:2},”a”]	list/dict
B	numeric array	f,1.2,13E-2,0	gfapy.NumericArray
H	byte array	FFAA01	gfapy.ByteArray

9.4 Validation

The tag names must consist of a letter and a digit or two letters.

```
"KC:i:1"  # => OK
"xx:i:1"  # => OK
"x1:i:1"  # => OK
"xxx:i:1" # => error: name is too long
"x:i:1"   # => error: name is too short
"11:i:1"  # => error: at least one letter must be present
```

The datatype must be one of the datatypes specified above. For predefined tags, Gfapy also checks that the datatype given in the specification is used.

```
"xx:X:1" # => error: datatype X is unknown
"VN:i:1" # => error: VN must be of type Z
```

The data must be a correctly formatted string for the specified datatype or a Python object whose string representation is a correctly formatted string.

```
# current value: xx:i:2
>>> line = gfapy.Line("S\tA\t*\txx:i:2")
>>> line.xx = 1
>>> line.xx
1
>>> line.xx = "3"
>>> line.xx
3
>>> line.xx = "A"
>>> line.xx
Traceback (most recent call last):
...
gfapy.error.FormatError: ...
```

Depending on the validation level, more or less checks are done automatically (see *Validation* chapter). Per default - validation level (1) - validation is performed only during parsing or accessing values the first time, therefore the user must perform a manual validation if he changes values to something which is not guaranteed to be correct. To trigger a manual validation, the user can call the method `validate_field(fieldname)` to validate a single tag, or `validate()` to validate the whole line, including all tags.

```
>>> line = gfapy.Line("S\tA\t*\txx:i:2", vlevel = 0)
>>> line.validate_field("xx")
>>> line.validate()
>>> line.xx = "A"
>>> line.validate_field("xx")
Traceback (most recent call last):
...
gfapy.error.FormatError: ...
>>> line.validate()
Traceback (most recent call last):
...
gfapy.error.FormatError: ...
>>> line.xx = "3"
>>> line.validate_field("xx")
>>> line.validate()
```

9.5 Reading and writing tags

Tags can be read using a property on the Gfapy line object, which is called as the tag (e.g. line.xx). A special version of the property prefixed by `try_get_` raises an error if the tag was not available (e.g. `line.try_get_LN`), while the tag property (e.g. `line.LN`) would return `None` in this case. Setting the value is done assigning a value to it the tag name method (e.g. `line.TS = 120`). In alternative, the `set(fieldname, value)`, `get(fieldname)` and `try_get(fieldname)` methods can also be used. To remove a tag from a line, use the `delete(fieldname)` method, or set its value to `None`. The `tagnames` property Line instances is a list of the names (as strings) of all defined tags for a line.

```
>>> line = gfapy.Line("S\tA\t*\txx:i:1", vlevel = 0)
>>> line.xx
1
>>> line.xy is None
True
>>> line.try_get_xx()
1
>>> line.try_get_xy()
Traceback (most recent call last):
...
gfapy.error.NotFoundError: ...
>>> line.get("xx")
1
>>> line.try_get("xy")
Traceback (most recent call last):
...
gfapy.error.NotFoundError: ...
>>> line.xx = 2
>>> line.xx
2
>>> line.xx = "a"
>>> line.tagnames
['xx']
>>> line.xy = 2
>>> line.xy
2
>>> line.set("xy", 3)
>>> line.get("xy")
3
>>> line.tagnames
['xx', 'xy']
>>> line.delete("xy")
3
>>> line.xy is None
True
>>> line.xx = None
>>> line.xx is None
True
>>> line.try_get("xx")
Traceback (most recent call last):
...
gfapy.error.NotFoundError: ...
>>> line.tagnames
[]
```

When a tag is read, the value is converted into an appropriate object (see Python classes in the datatype table above). When setting a value, the user can specify the value of a tag either as a Python object, or as the string representation of the value.

```
>>> line = gfapy.Line('H\txx:i:1\txy:Z:TEXT\txz:J:["a","b"]')
>>> line.xx
```

(continues on next page)

(continued from previous page)

```
1
>>> isinstance(line.xx, int)
True
>>> line.xy
'TEXT'
>>> isinstance(line.xy, str)
True
>>> line.xz
['a', 'b']
>>> isinstance(line.xz, list)
True
```

The string representation of a tag can be read using the `field_to_s(fieldname)` method. The default is to only output the content of the field. By setting ``tag: true```, the entire tag is output (name, datatype, content, separated by colons). An exception is raised if the field does not exist.

```
>>> line = gfapy.Line("H\txx:i:1")
>>> line.xx
1
>>> line.field_to_s("xx")
'1'
>>> line.field_to_s("xx", tag=True)
'xx:i:1'
```

9.6 Datatype of custom tags

The datatype of an existing custom field (but not of predefined fields) can be changed using the `set_datatype(fieldname, datatype)` method. The current datatype specification can be read using `get_datatype(fieldname)`.

```
>>> line = gfapy.Line("H\txx:i:1")
>>> line.get_datatype("xx")
'i'
>>> line.set_datatype("xx", "Z")
>>> line.get_datatype("xx")
'Z'
```

If a new custom tag is specified, Gfapy selects the correct datatype for it: i/f for numeric values, J/B for arrays, J for hashes and Z for strings and strings. If the user wants to specify a different datatype, he may do so by setting it with `set_datatype()` (this can be done also before assigning a value, which is necessary if full validation is active).

```
>>> line = gfapy.Line("H")
>>> line.xx = "1"
>>> line.xx
'1'
>>> line.set_datatype("xy", "i")
>>> line.xy = "1"
>>> line.xy
1
```

9.7 Arrays of numerical values

`B` and `H` tags represent array with particular constraints (e.g. they can only contain numeric values, and in some cases the values must be in predefined ranges). In order to represent them correctly and allow for validation, Python classes have been defined for both kind of tags: `gfapy.ByteArray` for `H` and `gfapy.NumericArray` for `B` fields.

Both are subclasses of list. Object of the two classes can be created by passing an existing list or the string representation to the class constructor.

```
>>> # create a byte array instance
>>> gfapy.ByteArray([12,3,14])
b'\x0c\x03\x0e'
>>> gfapy.ByteArray("A012FF")
b'\xa0\x12\xff'
>>> # create a numeric array instance
>>> gfapy.NumericArray.from_string("c,12,3,14")
[12, 3, 14]
>>> gfapy.NumericArray([12,3,14])
[12, 3, 14]
```

Instances of the classes behave as normal lists, except that they provide a #validate() method, which checks the constraints, and that their string representation is the GFA string representation of the field value.

```
>>> gfapy.NumericArray([12,1,"1x"]).validate()
Traceback (most recent call last):
...
gfapy.error.ValueError
>>> str(gfapy.NumericArray([12,3,14]))
'C,12,3,14'
>>> gfapy.ByteArray([12,1,"1x"]).validate()
Traceback (most recent call last):
...
gfapy.error.ValueError
>>> str(gfapy.ByteArray([12,3,14]))
'0C030E'
```

For numeric values, the `compute_subtype` method allows to compute the subtype which will be used for the string representation. Unsigned subtypes are used if all values are positive. The smallest possible subtype range is selected. The subtype may change when the range of the elements changes.

```
>>> gfapy.NumericArray([12,13,14]).compute_subtype()
'C'
```

9.8 Special cases: custom records, headers, comments and virtual lines.

GFA2 allows custom records, introduced by record type strings other than the predefined ones. Gfapy uses a pragmatical approach for identifying tags in custom records, and tries to interpret the rightmost fields as tags, until the first field from the right raises an error; all remaining fields are treated as positional fields.

```
"X a b c xx:i:12" # => xx is tag, a, b, c are positional fields
"Y a b xx:i:12 c" # => all positional fields, as c is not a valid tag
```

For easier access, the entire header of the GFA is summarized in a single line instance. A class (`FieldArray`) has been defined to handle the special case when multiple H lines define the same tag (see *The Header* chapter for details).

Comment lines are represented by a subclass of the same class (`Line`) as the records. However, they cannot contain tags: the entire line is taken as content of the comment. See the *Comments* chapter for more information about comments.

```
"# this is not a tag: xx:i:1" # => xx is not a tag, xx:i:1 is part of the comment
```

Virtual instances of the `Line` class (e.g. segment instances automatically created because of not yet resolved references found in edges) cannot be modified by the user, and tags cannot be specified for them. This includes all instances of the `Unknown` class. See the *References* chapter for more information about virtual lines.

CHAPTER

TEN

REFERENCES

Some fields in GFA lines contain identifiers or lists of identifiers (sometimes followed by orientation strings), which reference other lines of the GFA file. In Gfapy it is possible to follow these references and traverse the graph.

10.1 Connecting a line to a Gfa object

In stand-alone line instances, the identifiers which reference other lines are either strings containing the line name, pairs of strings (name and orientation) in a `gfapy.OrientedLine` object, or lists of lines names or `gfapy.OrientedLine` objects.

Using the `add_line(line)` (alias: `append(line)`) method of the `gfapy.Gfa` object, or the equivalent `connect(gfa)` method of the gfapy.Line instance, a line is added to a Gfa instance (this is done automatically when a GFA file is parsed). All strings expressing references are then changed into references to the corresponding line objects. The method `is_connected()` allows to determine if a line is connected to a gfapy instance. The read-only property `gfa` contains the `gfapy.Gfa` instance to which the line is connected.

```
>>> gfa = gfapy.Gfa(version='gfa1')
>>> link = gfapy.Line("L\tA\t-\tB\t+\t20M")
>>> link.is_connected()
False
>>> link.gfa is None
True
>>> type(link.from_segment)
<class 'str'>
>>> gfa.append(link)
>>> link.is_connected()
True
>>> link.gfa
<gfapy.gfa.Gfa object at ...>
>>> type(link.from_segment)
<class 'gfapy.line.segment.gfa1.GFA1'>
```

10.2 References for each record type

The following tables describes the references contained in each record type. The notation `[]` represent lists.

10.2.1 GFA1

Record type	Fields	Type of reference
Link	from, to	Segment
Containment	from, to	Segment
Path	segment_names,	[OrientedLine(Segment)]
	links (1)	[OrientedLine(Link)]

(1): paths contain information in the fields segment_names and overlaps, which allow to find the identify from which they depend; these links can be retrieved using `links` (which is not a field).

10.2.2 GFA2

Record type	Fields	Type of reference
Edge	sid1, sid2	Segment
Gap	sid1, sid2	Segment
Fragment	sid	Segment
Set	items	[Edge/Set/Path/Segment]
Path	items	[OrientedLine(Edge/Set/Segment)]

10.3 Backreferences for each record type

When a line containing a reference to another line is connected to a Gfa object, backreferences to it are created in the targeted line.

For each backreference collection a read-only property exist, which is named as the collection (e.g. `dovetails_L` for segments). Note that the reference list returned by these arrays are read-only and editing the references is done using other methods (see the section "Editing reference fields" below).

```
segment.dovetails_L # => [gfapy.line.edge.Link(...), ...]
```

The following tables describe the backreferences collections for each record type.

10.3.1 GFA1

Record type	Backreferences
Segment	dovetails_L
	dovetails_R
	edges_to_contained
	edges_to_containers
	paths
Link	paths

10.3.2 GFA2

Record type	Backreferences	Type
Segment	dovetails_L	E
	dovetails_R	E
	edges_to_contained	E
	edges_to_containers	E
	internals	E
	gaps_L	G
	gaps_R	G
	fragments	F
	paths	O
	sets	U
Edge	paths	O
	sets	U
O Group	paths	O
	sets	U
U Group	sets	U

10.3.3 Segment backreference convenience methods

For segments, additional methods are available which combine in different way the backreferences information. The `dovetails_of_end` and `gaps_of_end` methods take an argument `L` or `R` and return the dovetails overlaps (or gaps) of the left or, respectively, right end of the segment sequence (equivalent to the segment properties `dovetails_L`/`dovetails_R` and `gaps_L`/`gaps_R`).

The segment `containments` property is a list of both containments where the segment is the container or the contained segment. The segment `edges` property is a list of all edges (dovetails, containments and internals) with a reference to the segment.

Other methods directly compute list of segments from the edges lists mentioned above. The `neighbours_L`, `neighbours_R` properties and the `neighbours` method compute the set of segment instances which are connected by dovetails to the segment. The `containers` and `contained` properties similarly compute the set of segment instances which, respectively, contains the segment, or are contained in the segment.

```
>>> gfa = gfapy.Gfa()
>>> gfa.append('S\tA\t*')
>>> s = gfa.segment('A')
>>> gfa.append('S\tB\t*')
>>> gfa.append('S\tC\t*')
>>> gfa.append('L\tA\t-\tB\t+\t*')
>>> gfa.append('C\tA\t+\tC\t+\t10\t*')
>>> [str(l) for l in s.dovetails_of_end("L")]
['L\tA\t-\tB\t+\t*']
>>> s.dovetails_L == s.dovetails_of_end("L")
True
>>> s.gaps_of_end("R")
[]
>>> [str(e) for e in s.edges]
['L\tA\t-\tB\t+\t*', 'C\tA\t+\tC\t+\t10\t*']
>>> [str(n) for n in s.neighbours_L]
['S\tB\t*']
>>> s.containers
[]
>>> [str(c) for c in s.contained]
['S\tC\t*']
```

10.4 Multiline group definitions

The GFA2 specification opens the possibility (experimental) to define groups on multiple lines, by using the same ID for each line defining the group. This is supported by gfapy.

This means that if multiple `Ordered` or `Unordered` instances connected to a Gfa object have the same `gid`, they are merged into a single instance (technically the last one getting added to the graph object). The items list are merged.

The tags of multiple line defining a group shall not contradict each other (i.e. either are the tag names on different lines defining the group all different, or, if the same tag is present on different lines, the value and datatype must be the same, in which case the multiple definition will be ignored).

```
>>> gfa = gfapy.Gfa()
>>> gfa.add_line("U\tu1\ts1 s2 s3")
>>> [s.name for s in gfa.sets[-1].items]
['s1', 's2', 's3']
>>> gfa.add_line('U\tu1\t4 5')
>>> [s.name for s in gfa.sets[-1].items]
['s1', 's2', 's3', '4', '5']
```

10.5 Induced set and captured path

The item list in GFA2 sets and paths may not contain elements which are implicitly involved. For example a path may contain segments, without specifying the edges connecting them, if there is only one such edge. Alternatively a path may contain edges, without explicitly indicating the segments. Similarly a set may contain edges, but not the segments referred to in them, or contain segments which are connected by edges, without the edges themselves. Furthermore groups may refer to other groups (set to sets or paths, paths to paths only), which then indirectly contain references to segments and edges.

Gfapy provides methods for the computation of the sets of segments and edges which are implied by an ordered or unordered group. Thereby all references to subgroups are resolved and implicit elements are added, as described in the specification. The computation can, therefore, only be applied to connected lines. For unordered groups, this computation is provided by the method `induced_set()`, which returns an array of segment and edge instances. For ordered group, the computation is provided by the method `captured_path()`, which returns a list of `gfapy.OrientedLine` instances, alternating segment and edge instances (and starting and ending in segments).

The methods `induced_segments_set()`, `induced_edges_set()`, `captured_segments()` and `captured_edges()` return, respectively, the list of only segments or edges, in ordered or unordered groups.

```
>>> gfa = gfapy.Gfa()
>>> gfa.add_line("S\ts1\t100\t*")
>>> gfa.add_line("S\ts2\t100\t*")
>>> gfa.add_line("S\ts3\t100\t*")
>>> gfa.add_line("E\te1\ts1+\ts2-\t0\t10\t90\t100$\t*")
>>> gfa.add_line("U\tu1\ts1 s2 s3")
>>> u = gfa.sets[-1]
>>> [l.name for l in u.induced_edges_set]
['e1']
>>> [l.name for l in u.induced_segments_set ]
['s1', 's2', 's3']
>>> [l.name for l in u.induced_set ]
['s1', 's2', 's3', 'e1']
```

10.6 Disconnecting a line from a Gfa object

Lines can be disconnected using the `rm(line)` method of the `gfapy.Gfa` object or the `disconnect()` method of the line instance.

```
>>> gfa = gfapy.Gfa()
>>> gfa.append('S\tsA\t*')
>>> gfa.append('S\tsB\t*')
>>> line = gfa.segment("sA")
>>> gfa.segment_names
['sA', 'sB']
>>> gfa.rm(line)
>>> gfa.segment_names
['sB']
>>> line = gfa.segment('sB')
>>> line.disconnect()
>>> gfa.segment_names
[]
```

Disconnecting a line affects other lines as well. Lines which are dependent on the disconnected line are disconnected as well. Any other reference to disconnected lines is removed as well. In the disconnected line, references to lines are transformed back to strings and backreferences are deleted.

The following tables show which dependent lines are disconnected if they refer to a line which is being disconnected.

10.6.1 GFA1

Record type	Dependent lines
Segment	links (+ paths), containments
Link	paths

10.6.2 GFA2

Record type	Dependent lines
Segment	edges, gaps, fragments, sets, paths
Edge	sets, paths
Sets	sets, paths

10.7 Editing reference fields

In connected line instances, it is not allowed to directly change the content of fields containing references to other lines, as this would make the state of the Gfa object invalid.

Besides the fields containing references, some other fields are read-only in connected lines. Changing some of the fields would require moving the backreferences to other collections (position fields of edges and gaps, `from_orient` and `to_orient` of links). The overlaps field of connected links is readonly as it may be necessary to identify the link in paths.

10.7.1 Renaming an element

The name field of a line (e.g. segment `name`/`sid`) is not a reference and thus can be edited also in connected lines. When the name of the line is changed, no manual editing of references (e.g. from/to fields in links) is necessary, as all lines which refer to the line will still refer to the same instance. The references to the instance in the Gfa lines collections will be automatically updated. Also, the new name will be correctly used when converting to string, such as when the Gfa instance is written to a GFA file.

Renaming a line to a name which already exists has the same effect of adding a line with that name. That is, in most cases, `gfapy.NotUniqueError` is raised. An exception are GFA2 sets and paths: in this case the line will be appended to the existing line with the same name (as described in "Multiline group definitions").

10.7.2 Adding and removing group elements

Elements of GFA2 groups can be added and removed from both connected and non-connected lines, using the following methods.

To add an item to or remove an item from an unordered group, use the methods `add_item(item)` and `rm_item(item)`, which take as argument either a string (identifier) or a line instance.

To append or prepend an item to an ordered group, use the methods `append_item(item)` and `prepend_item(item)`. To remove the first or the last item of an ordered group use the methods `rm_first_item()` and `rm_last_item()`.

10.7.3 Editing read-only fields of connected lines

Editing the read-only information of edges, gaps, links, containments, fragments and paths is more complicated. These lines shall be disconnected before the edit and connected again to the Gfa object after it. Before disconnecting a line, you should check if there are other lines dependent on it (see tables above). If so, you will have to disconnect these lines first, eventually update their fields and reconnect them at the end of the operation.

10.8 Virtual lines

The order of the lines in GFA is not prescribed. Therefore, during parsing, or constructing a Gfa in memory, it is possible that a line is referenced to, before it is added to the Gfa instance. Whenever this happens, Gfapy creates a "virtual" line instance.

Users do not have to handle with virtual lines, if they work with complete and valid GFA files.

Virtual lines are similar to normal line instances, with some limitations (they contain only limited information and it is not allowed to add tags to them). To check if a line is a virtual line, one can use the `virtual` property of the line.

As soon as the parser founds the real line corresponding to a previously introduced virtual line, the virtual line is exchanged with the real line and all references are corrected to point to the real line.

```
>>> g = gfapy.Gfa()
>>> g.add_line("S\t1\t*")
>>> g.add_line("L\t1\t+\t2\t+\t*")
>>> l = g.dovetails[0]
>>> g.segment("1").virtual
False
>>> g.segment("2").virtual
True
>>> l.to_segment == g.segment("2")
True
>>> g.segment("2").dovetails == [l]
True
```

(continues on next page)

(continued from previous page)

```
>>> g.add_line("S\t2\t*")
>>> g.segment("2").virtual
False
>>> l.to_segment == g.segment("2")
True
>>> g.segment("2").dovetails == [l]
True
```

CHAPTER

ELEVEN

THE HEADER

GFA files may contain one or multiple header lines (record type: "H"). These lines may be present in any part of the file, not necessarily at the beginning.

Although the header may consist of multiple lines, its content refers to the whole file. Therefore in Gfapy the header is accessed using a single line instance (accessible by the `header` property). Header lines contain only tags. If not header line is present in the Gfa, then the header line object will be empty (i.e. contain no tags).

Note that header lines cannot be connected to the Gfa as other lines (i.e. calling `connect()` on them raises an exception). Instead they must be merged to the existing Gfa header, using `add_line` on the Gfa instance.

```
>>> gfa.add_line("H\tnn:f:1.0")
>>> gfa.header.nn
1.0
>>> gfapy.Line("H\tnn:f:1.0").connect(gfa)
Traceback (most recent call last):
...
gfapy.error.RuntimeError: ...
```

11.1 Multiple definitions of the predefined header tags

For the predefined tags (`VN` and `TS`), the presence of multiple values in different lines is an error, unless the value is the same in each instance (in which case the repeated definitions are ignored).

```
>>> gfa.add_line("H\tVN:Z:1.0")
>>> gfa.add_line("H\tVN:Z:1.0") # ignored
>>> gfa.add_line("H\tVN:Z:2.0")
Traceback (most recent call last):
...
gfapy.error.VersionError: ...
```

11.2 Multiple definitions of custom header tags

If the tags are present only once in the header in its entirety, the access to the tags is the same as for any other line (see the *Tags* chapter).

However, the specification does not forbid custom tags to be defined with different values in different header lines (which we name "multi-definition tags"). This particular case is handled in the next sections.

11.3 Reading multi-definitions tags

Reading, validating and setting the datatype of multi-definition tags is done using the same methods as for all other lines (see the *Tags* chapter). However, if a tag is defined multiple times on multiple H lines, reading the tag will return a list of the values on the lines. This array is an instance of the subclass `gfapy.FieldArray` of list.

```
>>> gfa.add_line("H\txx:i:1")
>>> gfa.add_line("H\txx:i:2")
>>> gfa.add_line("H\txx:i:3")
>>> gfa.header.xx
gfapy.FieldArray('i',[1, 2, 3])
```

11.4 Setting tags

There are two possibilities to set a tag for the header. The first is the normal tag interface (using `set` or the tag name property). The second is to use `add`. The latter supports multi-definition tags, i.e. it adds the value to the previous ones (if any), instead of overwriting them.

```
>>> gfa = gfapy.Gfa()
>>> gfa.header.xx
>>> gfa.header.add("xx", 1)
>>> gfa.header.xx
1
>>> gfa.header.add("xx", 2)
>>> gfa.header.xx
gfapy.FieldArray('i',[1, 2])
>>> gfa.header.set("xx", 3)
>>> gfa.header.xx
3
```

11.5 Modifying field array values

Field arrays can be modified directly (e.g. adding new values or removing some values). After modification, the user may check if the array values remain compatible with the datatype of the tag using the `validate_field`` ()` method.

```
>>> gfa = gfapy.Gfa()
>>> gfa.header.xx = gfapy.FieldArray('i',[1,2,3])
>>> gfa.header.xx
gfapy.FieldArray('i',[1, 2, 3])
>>> gfa.header.validate_field("xx")
>>> gfa.header.xx.append("X")
>>> gfa.header.validate_field("xx")
Traceback (most recent call last):
...
gfapy.error.FormatError: ...
```

If the field array is modified using array methods which return a list or data of any other type, a field array must be constructed, setting its datatype to the value returned by calling `get_datatype()` on the header.

```
>>> gfa = gfapy.Gfa()
>>> gfa.header.xx = gfapy.FieldArray('i',[1,2,3])
>>> gfa.header.xx
gfapy.FieldArray('i',[1, 2, 3])
>>> gfa.header.xx = gfapy.FieldArray(gfa.header.get_datatype("xx"),
... list(map(lambda x: x+1, gfa.header.xx)))
```

(continues on next page)

(continued from previous page)

```
>>> gfa.header.xx
gfapy.FieldArray('i',[2, 3, 4])
```

11.6 String representation of the header

For consistency with other line types, the string representation of the header is a single-line string, eventually non standard-compliant, if it contains multiple instances of the tag. (and when calling `field_to_s()` for a tag present multiple times, the output string will contain the instances of the tag, separated by tabs).

However, when the Gfa is output to file or string, the header is split into multiple H lines with single tags, so that standard-compliant GFA is output. The split header can be retrieved using the `headers` property of the Gfa instance.

```
>>> gfa = gfapy.Gfa()
>>> gfa.header.VN = "1.0"
>>> gfa.header.xx = gfapy.FieldArray('i',[1,2])
>>> gfa.header.field_to_s("xx")
'1\t2'
>>> gfa.header.field_to_s("xx", tag=True)
'xx:i:1\txx:i:2'
>>> str(gfa.header)
'H\tVN:Z:1.0\txx:i:1\txx:i:2'
>>> [str(h) for h in gfa.headers]
['H\tVN:Z:1.0', 'H\txx:i:1', 'H\txx:i:2']
>>> str(gfa)
'H\tVN:Z:1.0\nH\txx:i:1\nH\txx:i:2'
```

CHAPTER

TWELVE

CUSTOM RECORDS

The GFA2 specification considers each line which starts with a non-standard record type a custom (i.e. user- or program-specific) record. Gfapy allows to retrieve these records and access their data using a similar interface to that for the predefined record types.

12.1 Retrieving, adding and deleting custom records

Gfa instances have the property `custom_records()`, a list of all line instances with a non-standard record type. Among these, records of a specific record type are retrieved using the method `Gfa.custom_records_of_type(record_type)`. Lines are added and deleted using the same methods (`add_line()` and `disconnect()`) as for other line types.

```
>>> g.add_line("X\tcustom line")
>>> g.add_line("Y\tcustom line")
>>> [str(line) for line in g.custom_records]
['X\tcustom line', 'Y\tcustom line']
>>> g.custom_record_keys)
['X', 'Y']
>>> [str(line) for line in g.custom_records_of_type('X')]
['X\tcustom line']
>>> g.custom_records_of_type("X")[-1].disconnect()
>>> g.custom_records_of_type('X')
[]
```

12.2 Interface without extensions

If no extension (see *Extensions* section) has been defined to handle a custom record type, the interface has some limitations: the field content is not validated, and the field names are unknown. The generic custom record class is employed (`CustomRecord`).

As the name of the positional fields in a custom record is not known, a generic name `field1`, `field2`, ... is used. The number of positional fields is found by getting the length of the `positional_fieldnames` list.

```
>>> g.add_line("X\ta\tb\tcc:i:10\tdd:i:100")
>>> x = g.custom_records_of_type('X')[-1]
>>> len(x.positional_fieldnames)
2
>>> x.field1
'a'
>>> x.field2
'b'
```

Positional fields are allowed to contain any character (including non-printable characters and spacing characters), except tabs and newlines (as they are structural elements of the line). No further validation is performed.

As Gfapy cannot know how many positional fields are present when parsing custom records, a heuristic approach is followed, to identify tags. A field resembles a tag if it starts with `tn:d:` where `tn` is a valid tag name and `d` a valid tag datatype (see *Tags* chapter). The fields are parsed from the last to the first.

As soon as a field is found which does not resemble a tag, all remaining fields are considered positionals (even if another field parsed later resembles a tag). Due to this, invalid tags are sometimes wrongly taken as positional fields (this can be avoided by writing an extension).

```
>>> g.add_line("X\ta\tb\tcc:i:10\tdd:i:100")
>>> x1 = g.custom_records_of_type("X")[-1]
>>> x1.cc
10
>>> x1.dd
100
>>> g.add_line("X\ta\tb\tcc:i:10\tdd:i:100\te")
>>> x2 = g.custom_records_of_type("X")[-1]
>>> x2.cc
>>> x2.field3
'cc:i:10'
>>> g.add_line("Z\ta\tb\tcc:i:10\tddd:i:100")
>>> x3 = g.custom_records_of_type("Z")[-1]
>>> x3.cc
>>> x3.field3
'cc:i:10'
>>> x3.field4
'ddd:i:100'
```

12.3 Extensions

The support for custom fields is limited, as Gfapy does not know which and how many fields are there and how shall they be validated. It is possible to create an extension of Gfapy, which defines new record types: this will allow to use these record types in a similar way to the built-in types.

As an example, an extension will be described, which defines two record types: T for taxa and M for assignments of segments to taxa. For further information about the possible usage case for this extension, see the Supplemental Information to the manuscript describing Gfapy.

The T records will contain a single positional field, `tid`, a GFA2 identifier, and an optional UL string tag. The M records will contain three positional fields (all three GFA2 identifier): a name field `mid` (optional), and two references, `tid` to a T line and `sid` to an S line. The SC integer tag will be also defined. Here is an example of a GFA containing M and T lines:

```
S sA 1000 *
S sB 1000 *
M assignment1 t123 sA SC:i:40
M assignment2 t123 sB
M * B12c sB SC:i:20
T B12c
T t123 UL:Z:http://www.taxon123.com
```

Writing subclasses of the `Line` class, it is possible to communicate to Gfapy, how records of the M and T class shall be handled. This only requires to define some constants and to call the class method `register_extension()`.

The constants to define are `RECORD_TYPE`, which shall be the content of the record type field (e.g. `M`); `POSFIELDS` shall contain an ordered dict, specifying the datatype for each positional field, in the order these fields are found in the line; `TAGS_DATATYPE` is a dict, specifying the datatype of the predefined optional tags; `NAME_FIELD` is a field name, and specifies which field contains the identifier of the line. For details on predefined and custom datatypes, see the next sections (*Predefined datatypes for extensions* and *Custom datatypes for extensions*).

To handle references, `register_extension()` can be supplied with a `references` parameter, a list of triples `(fieldname, classname, backreferences)`. Thereby `fieldname` is the name of the field in the corresponding record containing the reference (e.g. `sid`), `classname` is the name of the class to which the reference goes (e.g. `gfa.line.segment.GFA2`), and texttt{backreferences} is how the collection of backreferences shall be called, in the records to which reference points to (e.g. `metagenomic_assignments`).

```
from collections include OrderedDict

class Taxon(gfapy.Line):
  RECORD_TYPE = "T"
  POSFIELDS = OrderedDict([("tid","identifier_gfa2")])
  TAGS_DATATYPE = {"UL":"Z"}
  NAME_FIELD = "tid"

Taxon.register_extension()

class MetagenomicAssignment(gfapy.Line):
  RECORD_TYPE = "M"
  POSFIELDS = OrderedDict([("mid","optional_identifier_gfa2"),
                           ("tid","identifier_gfa2"),
                           ("sid","identifier_gfa2")])
  TAGS_DATATYPE = {"SC":"i"}
  NAME_FIELD = "mid"

MetagenomicAssignment.register_extension(references=
    [("sid", gfapy.line.segment.GFA2, "metagenomic_assignments"),
     ("tid", Taxon, "metagenomic_assignments")])
```

12.4 Predefined datatypes for extensions

The datatype of fields is specified in Gfapy using classes, which provide functions for decoding, encoding and validating the corresponding data. Gfapy contains a number of datatypes which correspond to the description of the field content in the GFA1 and GFA2 specification.

When writing extensions only the GFA2 field datatypes are generally used (as GFA1 does not contain custom fields). They are summarized in the following table:

Name	Example	Description
`alignment_gfa2`	`12M1I3M`	CIGAR string, Trace alignment or Placeholder (`*`)
`identifier_gfa2`	`S1`	ID of a line
`oriented_identifier_gfa2`	`S1+`	ID of a line followed by + or −
`optional_identifier_gfa2`	`*`	ID of a line or Placeholder (`*`)
`identifier_list_gfa2`	`S1 S2`	space separated list of line IDs
`oriented_identifier_list_gfa2`	`S1+ S2-`	space separated list of line IDs plus orientations
`position_gfa2`	`120$`	non-negative integer, optionally followed by `$`
`sequence_gfa2`	`ACGNNYR`	sequence of printable chars., no whitespace
`string`	`a b_c; d`	string, no tabs and newlines (Z tags)
`char`	`A`	single character (A tags)
`float`	`1.12`	float (f tags)
`integer`	`-12`	integer (i tags)
`optional_integer`	`*`	integer or placeholder
`numeric_array`	`c,10,3`	array of integers or floats (B tags)
`byte_array`	`12F1FF`	hexadecimal byte string (H tags)
`json`	`{'b':2}`	JSON string, no tabs and newlines (J tags)

12.5 Custom datatypes for extensions

For custom records, one sometimes needs datatypes not yet available in the GFA specification. For example, a custom datatype can be defined for the taxon identifier used in the `tid` field of the T and M records: accordingly the taxon identifier shall be only either in the form `taxon:<n>`, where `<n>` is a positive integer, or consist of letters, numbers and underscores only (without `:`).

To define the datatype, a class is written, which contains the following functions:

- `validate_encoded(string)`: validates the content of the field, if this is a string (e.g., the name of the T line)
- `validate_decoded(object)`: validates the content of the field, if this is not a string (e.g., a reference to a T line)
- `decode(string)`: validates the content of the field (a string) and returns the decoded content; note that references must not be resolved (there is no access to the Gfa instance here), thus the name of the T line will be returned unchanged
- `encode(string)`: validates the content of the field (not in string form) and returns the string which codes it in the GFA file (also here references are validated but not converted into strings)

Finally the datatype is registered calling `register_datatype()`. The code for the taxon ID extension is the following:

```
import re

class TaxonID:

  def validate_encoded(string):
    if not re.match(r"^taxon:(\d+)$",string) and \
        not re.match(r"^[a-zA-Z0-9_]+$", string):
      raise gfapy.ValueError("Invalid taxon ID: {}".format(string))

  def decode(string):
    TaxonID.validate_encoded(string)
    return string

  def validate_decoded(obj):
    if isinstance(obj,Taxon):
      TaxonID.validate_encoded(obj.name)
    else:
      raise gfapy.TypeError(
        "Invalid type for taxon ID: "+"{}".format(repr(obj)))

  def encode(obj):
    TaxonID.validate_decoded(obj)
    return obj

gfapy.Field.register_datatype("taxon_id", TaxonID)
```

To use the new datatype in the T and M lines defined above (*Extensions*), the definition of the two subclasses can be changed: in `POSFIELDS` the value `taxon_id` shall be assigned to the key `tid`.

CHAPTER

THIRTEEN

COMMENTS

GFA lines starting with a # symbol are considered comments. In Gfapy comments are represented by instances of the class `gfapy.line.Comment`. They have a similar interface to other line instances, with some differences, e.g. they do not support tags.

13.1 The comments collection

The comments of a Gfa object are accessed using the `Gfa.comments` property. This is a list of comment line instances. The single elements can be modified, but the list itself is read-only. To remove a comment from the Gfa, you need to find the instance in the list, and call `disconnect()` on it. To add a comment to a `Gfa` instance is done similarly to other lines, by using the `Gfa.add_line(line)` method.

```
>>> g.add_line("# this is a comment")
>>> [str(c) for c in g.comments]
['# this is a comment']
>>> g.comments[0].disconnect()
>>> g.comments
[]
```

13.2 Accessing the comment content

The content of the comment line, excluding the initial # and eventual initial spacing characters, is included in the `content` field. The initial spacing characters can be read/changed using the `spacer` field. The default value is a single space.

```
>>> g.add_line("# this is a comment")
>>> c = g.comments[-1]
>>> c.content
'this is a comment'
>>> c.spacer
' '
>>> c.spacer = '___'
>>> str(c)
'#___this is a comment'
```

Tags are not supported by comment lines. If the line contains tags, these are nor parsed, but included in the `content` field. Trying to set tags raises exceptions.

```
>>> c = gfapy.Line("# this is not a tag\txx:i:1")
>>> c.content
'this is not a tag\txx:i:1'
>>> c.xx
>>> c.xx = 1
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
gfapy.error.RuntimeError: Tags of comment lines cannot be set
```

CHAPTER

FOURTEEN

ERRORS

The different types of errors defined in Gfapy are summarized in the following table. All exception raised in the library are subclasses of `Error`. Thus, `except gfapy.Error` can be use to catch all library errors.

Error	Description	Examples
`VersionError`	An unknown or wrong version is specified or implied	"GFA0"; or GFA1 in GFA2 context
`ValueError`	The value of an object is invalid	a negative position is used
`TypeError`	The wrong type has been used or specified	Z instead of i used for VN tag; Hash for an i tag
`FormatError`	The format of an object is wrong	a line does not contain the expected number of fields
`NotUniqueError`	Something should be unique but is not	duplicated tag name or line identifier
`InconsistencyError`	Pieces of information collide with each other	length of sequence and LN tag do not match
`RuntimeError`	The user tried to do something which is not allowed	editing from/to field in connected links
`ArgumentError`	Problem with the arguments of a method	wrong number of arguments in dynamically created method
`AssertionError`	Something unexpected happened	there is a bug in the library or the library has been used in an unintended way

CHAPTER

FIFTEEN

GRAPH OPERATIONS

Graph operations such as linear paths merging, multiplication of segments and other are provided. These operations are implemented in analogy to those provided by the Ruby library RGFA. As RGFA only handles GFA1 graphs, only dovetail overlaps are considered as connections. A detailed description of the operation can be found in Gonnella and Kurtz (2016). More information about the single operations are found in the method documentation of the submodules of `GraphOperations`.

CHAPTER

SIXTEEN

RGFA

rGFA (https://github.com/lh3/gfatools/blob/master/doc/rGFA.md) is a subset of GFA1, in which only particular line types (S and L) are allowed, and the S lines are required to contain the tags `SN` (of type `Z`), `SO` and `SR` (of type `i`).

When working with rGFA files, it is convenient to use the `dialect="rgfa"` option in the constructor `Gfa()` and in func:`Gfa.from_file()`.

This ensures that additional validations are performed: GFA version must be 1, only rGFA-compatible lines (S,L) are allowed and that the required tags are required (with the correct datatype). The validations can also be executed manually using `Gfa.validate_rgfa()`.

Furthermore, the `stable_sequence_names` attribute of the GFA objects returns the set of stable sequence names contained in the `SN` tags of the segments.

```
>>> g = gfapy.Gfa("S\tS1\tCTGAA\tSN:Z:chr1\tSO:i:0\tSR:i:0", dialect="rgfa")
>>> g.segment_names
['S1']
>>> g.stable_sequence_names
['chr1']
>>> g.add_line("S\tS2\tACG\tSN:Z:chr1\tSO:i:5\tSR:i:0")
```

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search