- The output file may contain results from one or more search queries.
- In each search query, you will see one or more hits from the given search database.
- In each database hit, you will see one or more regions containing the actual sequence alignment between your query sequence and the database sequence.
- Some programs like BLAT or Exonerate may further split these regions into several alignment fragments (or blocks in BLAT and possibly exons in exonerate). This is not something you always see, as programs like BLAST and HMMER do not do this.

Realizing this generality, we decided use it as base for creating the `Bio.SearchIO` object model. The object model consists of a nested hierarchy of Python objects, each one representing one concept outlined above. These objects are:

- `QueryResult`, to represent a single search query.
- `Hit`, to represent a single database hit. `Hit` objects are contained within `QueryResult` and in each `QueryResult` there is zero or more `Hit` objects.
- `HSP` (short for high-scoring pair), to represent region(s) of significant alignments between query and hit sequences. `HSP` objects are contained within `Hit` objects and each `Hit` has one or more `HSP` objects.
- `HSPFragment`, to represent a single contiguous alignment between query and hit sequences. `HSPFragment` objects are contained within `HSP` objects. Most sequence search tools like BLAST and HMMER unify `HSP` and `HSPFragment` objects as each `HSP` will only have a single `HSPFragment`. However there are tools like BLAT and Exonerate that produce `HSP` containing multiple `HSPFragment`. Don't worry if this seems a tad confusing now, we'll elaborate more on these two objects later on.

These four objects are the ones you will interact with when you use `Bio.SearchIO`. They are created using one of the main `Bio.SearchIO` methods: `read`, `parse`, `index`, or `index_db`. The details of these methods are provided in later sections. For this section, we'll only be using read and parse. These functions behave similarly to their `Bio.SeqIO` and `Bio.AlignIO` counterparts:

- `read` is used for search output files with a single query and returns a `QueryResult` object
- `parse` is used for search output files with multiple queries and returns a generator that yields `QueryResult` objects

With that settled, let's start probing each `Bio.SearchIO` object, beginning with `QueryResult`.

11.1.1 QueryResult

The QueryResult object represents a single search query and contains zero or more Hit objects. Let's see what it looks like using the BLAST file we have:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> print(blast_qresult)
Program: blastn (2.2.27+)
  Query: 42291 (61)
         mystery_seq
 Target: refseq_rna
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
```

```
----  -----  ------------------------------------------------------------
   0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
   1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
   2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
   3      2  gi|301171322|ref|NR_035857.1|  Pan troglodytes microRNA...
   4      1  gi|301171267|ref|NR_035851.1|  Pan troglodytes microRNA...
   5      2  gi|262205330|ref|NR_030198.1|  Homo sapiens microRNA 52...
   6      1  gi|262205302|ref|NR_030191.1|  Homo sapiens microRNA 51...
   7      1  gi|301171259|ref|NR_035850.1|  Pan troglodytes microRNA...
   8      1  gi|262205451|ref|NR_030222.1|  Homo sapiens microRNA 51...
   9      2  gi|301171447|ref|NR_035871.1|  Pan troglodytes microRNA...
  10      1  gi|301171276|ref|NR_035852.1|  Pan troglodytes microRNA...
  11      1  gi|262205290|ref|NR_030188.1|  Homo sapiens microRNA 51...
  12      1  gi|301171354|ref|NR_035860.1|  Pan troglodytes microRNA...
  13      1  gi|262205281|ref|NR_030186.1|  Homo sapiens microRNA 52...
  14      2  gi|262205298|ref|NR_030190.1|  Homo sapiens microRNA 52...
  15      1  gi|301171394|ref|NR_035865.1|  Pan troglodytes microRNA...
  16      1  gi|262205429|ref|NR_030218.1|  Homo sapiens microRNA 51...
  17      1  gi|262205423|ref|NR_030217.1|  Homo sapiens microRNA 52...
  18      1  gi|301171401|ref|NR_035866.1|  Pan troglodytes microRNA...
  19      1  gi|270133247|ref|NR_032574.1|  Macaca mulatta microRNA ...
  20      1  gi|262205309|ref|NR_030193.1|  Homo sapiens microRNA 52...
  21      2  gi|270132717|ref|NR_032716.1|  Macaca mulatta microRNA ...
  22      2  gi|301171437|ref|NR_035870.1|  Pan troglodytes microRNA...
  23      2  gi|270133306|ref|NR_032587.1|  Macaca mulatta microRNA ...
  24      2  gi|301171428|ref|NR_035869.1|  Pan troglodytes microRNA...
  25      1  gi|301171211|ref|NR_035845.1|  Pan troglodytes microRNA...
  26      2  gi|301171153|ref|NR_035838.1|  Pan troglodytes microRNA...
  27      2  gi|301171146|ref|NR_035837.1|  Pan troglodytes microRNA...
  28      2  gi|270133254|ref|NR_032575.1|  Macaca mulatta microRNA ...
  29      2  gi|262205445|ref|NR_030221.1|  Homo sapiens microRNA 51...
  ~~~
  97      1  gi|356517317|ref|XM_003527287.1|  PREDICTED: Glycine ma...
  98      1  gi|297814701|ref|XM_002875188.1|  Arabidopsis lyrata su...
  99      1  gi|397513516|ref|XM_003827011.1|  PREDICTED: Pan panisc...
```

We've just begun to scratch the surface of the object model, but you can see that there's already some useful information. By invoking `print` on the `QueryResult` object, you can see:

- The program name and version (blastn version 2.2.27+)
- The query ID, description, and its sequence length (ID is 42291, description is 'mystery_seq', and it is 61 nucleotides long)
- The target database to search against (refseq_rna)
- A quick overview of the resulting hits. For our query sequence, there are 100 potential hits (numbered 0–99 in the table). For each hit, we can also see how many HSPs it contains, its ID, and a snippet of its description. Notice here that `Bio.SearchIO` truncates the hit table overview, by showing only hits numbered 0–29, and then 97–99.

Now let's check our BLAT results using the same procedure as above:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> print(blat_qresult)
Program: blat (<unknown version>)
  Query: mystery_seq (61)
         <unknown description>
 Target: <unknown target>
   Hits: ----  -----  ----------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ----------------------------------------------------------
            0     17  chr19  <unknown description>
```

You'll immediately notice that there are some differences. Some of these are caused by the way PSL format stores its details, as you'll see. The rest are caused by the genuine program and target database differences between our BLAST and BLAT searches:

- The program name and version. `Bio.SearchIO` knows that the program is BLAT, but in the output file there is no information regarding the program version so it defaults to '¡unknown version¿'.
- The query ID, description, and its sequence length. Notice here that these details are slightly different from the ones we saw in BLAST. The ID is 'mystery_seq' instead of 42991, there is no known description, but the query length is still 61. This is actually a difference introduced by the file formats themselves. BLAST sometimes creates its own query IDs and uses your original ID as the sequence description.
- The target database is not known, as it is not stated in the BLAT output file.
- And finally, the list of hits we have is completely different. Here, we see that our query sequence only hits the 'chr19' database entry, but in it we see 17 HSP regions. This should not be surprising however, given that we are using a different program, each with its own target database.

All the details you saw when invoking the `print` method can be accessed individually using Python's object attribute access notation (a.k.a. the dot notation). There are also other format-specific attributes that you can access using the same method.

```
>>> print("%s %s" % (blast_qresult.program, blast_qresult.version))
blastn 2.2.27+
>>> print("%s %s" % (blat_qresult.program, blat_qresult.version))
blat <unknown version>
>>> blast_qresult.param_evalue_threshold  # blast-xml specific
10.0
```

For a complete list of accessible attributes, you can check each format-specific documentation. Here are the ones for BLAST and for BLAT.

Having looked at using `print` on `QueryResult` objects, let's drill down deeper. What exactly is a `QueryResult`? In terms of Python objects, `QueryResult` is a hybrid between a list and a dictionary. In other words, it is a container object with all the convenient features of lists and dictionaries.

Like Python lists and dictionaries, `QueryResult` objects are iterable. Each iteration returns a `Hit` object:

```
>>> for hit in blast_qresult:
...     hit
...
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171311|ref|NR_035856.1|', query_id='42291', 1 hsps)
Hit(id='gi|270133242|ref|NR_032573.1|', query_id='42291', 1 hsps)
Hit(id='gi|301171322|ref|NR_035857.1|', query_id='42291', 2 hsps)
Hit(id='gi|301171267|ref|NR_035851.1|', query_id='42291', 1 hsps)
...
```

To check how many items (hits) a `QueryResult` has, you can simply invoke Python's `len` method:

```
>>> len(blast_qresult)
100
>>> len(blat_qresult)
1
```

Like Python lists, you can retrieve items (hits) from a `QueryResult` using the slice notation:

```
>>> blast_qresult[0]  # retrieves the top hit
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
>>> blast_qresult[-1]  # retrieves the last hit
Hit(id='gi|397513516|ref|XM_003827011.1|', query_id='42291', 1 hsps)
```

To retrieve multiple hits, you can slice `QueryResult` objects using the slice notation as well. In this case, the slice will return a new `QueryResult` object containing only the sliced hits:

```
>>> blast_slice = blast_qresult[:3]  # slices the first three hits
>>> print(blast_slice)
Program: blastn (2.2.27+)
  Query: 42291 (61)
         mystery_seq
 Target: refseq_rna
   Hits: ----  -----  ------------------------------------------------------------
            #  # HSP  ID + description
         ----  -----  ------------------------------------------------------------
            0      1  gi|262205317|ref|NR_030195.1|  Homo sapiens microRNA 52...
            1      1  gi|301171311|ref|NR_035856.1|  Pan troglodytes microRNA...
            2      1  gi|270133242|ref|NR_032573.1|  Macaca mulatta microRNA ...
```

Like Python dictionaries, you can also retrieve hits using the hit's ID. This is particularly useful if you know a given hit ID exists within a search query results:

```
>>> blast_qresult["gi|262205317|ref|NR_030195.1|"]
Hit(id='gi|262205317|ref|NR_030195.1|', query_id='42291', 1 hsps)
```

You can also get a full list of `Hit` objects using `hits` and a full list of `Hit` IDs using `hit_keys`:

```
>>> blast_qresult.hits
[...]       # list of all hits
>>> blast_qresult.hit_keys
[...]       # list of all hit IDs
```

What if you just want to check whether a particular hit is present in the query results? You can do a simple Python membership test using the `in` keyword:

```
>>> "gi|262205317|ref|NR_030195.1|" in blast_qresult
True
>>> "gi|262205317|ref|NR_030194.1|" in blast_qresult
False
```

Sometimes, knowing whether a hit is present is not enough; you also want to know the rank of the hit. Here, the `index` method comes to the rescue:

```
>>> blast_qresult.index("gi|301171437|ref|NR_035870.1|")
22
```

Remember that we're using Python's indexing style here, which is zero-based. This means our hit above is ranked at no. 23, not 22.

Also, note that the hit rank you see here is based on the native hit ordering present in the original search output file. Different search tools may order these hits based on different criteria.

If the native hit ordering doesn't suit your taste, you can use the `sort` method of the `QueryResult` object. It is very similar to Python's `list.sort` method, with the addition of an option to create a new sorted `QueryResult` object or not.

Here is an example of using `QueryResult.sort` to sort the hits based on each hit's full sequence length. For this particular sort, we'll set the `in_place` flag to `False` so that sorting will return a new `QueryResult` object and leave our initial object unsorted. We'll also set the `reverse` flag to `True` so that we sort in descending order.

```
>>> for hit in blast_qresult[:5]:  # id and sequence length of the first five hits
...     print("%s %i" % (hit.id, hit.seq_len))
...
gi|262205317|ref|NR_030195.1| 61
gi|301171311|ref|NR_035856.1| 60
gi|270133242|ref|NR_032573.1| 85
gi|301171322|ref|NR_035857.1| 86
gi|301171267|ref|NR_035851.1| 80

>>> sort_key = lambda hit: hit.seq_len
>>> sorted_qresult = blast_qresult.sort(key=sort_key, reverse=True, in_place=False)
>>> for hit in sorted_qresult[:5]:
...     print("%s %i" % (hit.id, hit.seq_len))
...
gi|397513516|ref|XM_003827011.1| 6002
gi|390332045|ref|XM_776818.2| 4082
gi|390332043|ref|XM_003723358.1| 4079
gi|356517317|ref|XM_003527287.1| 3251
gi|356543101|ref|XM_003539954.1| 2936
```

The advantage of having the `in_place` flag here is that we're preserving the native ordering, so we may use it again later. You should note that this is not the default behavior of `QueryResult.sort`, however, which is why we needed to set the `in_place` flag to `True` explicitly.

At this point, you've known enough about `QueryResult` objects to make it work for you. But before we go on to the next object in the `Bio.SearchIO` model, let's take a look at two more sets of methods that could make it even easier to work with `QueryResult` objects: the `filter` and `map` methods.

If you're familiar with Python's list comprehensions, generator expressions or the built in `filter` and `map` functions, you'll know how useful they are for working with list-like objects (if you're not, check them out!). You can use these built in methods to manipulate `QueryResult` objects, but you'll end up with regular Python lists and lose the ability to do more interesting manipulations.

That's why, `QueryResult` objects provide its own flavor of `filter` and `map` methods. Analogous to `filter`, there are `hit_filter` and `hsp_filter` methods. As their name implies, these methods filter its `QueryResult` object either on its `Hit` objects or `HSP` objects. Similarly, analogous to `map`, `QueryResult` objects also provide the `hit_map` and `hsp_map` methods. These methods apply a given function to all hits or HSPs in a `QueryResult` object, respectively.

Let's see these methods in action, beginning with `hit_filter`. This method accepts a callback function that checks whether a given `Hit` object passes the condition you set or not. In other words, the function must accept as its argument a single `Hit` object and returns `True` or `False`.

Here is an example of using `hit_filter` to filter out `Hit` objects that only have one HSP:

```
>>> filter_func = lambda hit: len(hit.hsps) > 1  # the callback function
>>> len(blast_qresult)  # no. of hits before filtering
100
>>> filtered_qresult = blast_qresult.hit_filter(filter_func)
>>> len(filtered_qresult)  # no. of hits after filtering
37
>>> for hit in filtered_qresult[:5]:  # quick check for the hit lengths
...     print("%s %i" % (hit.id, len(hit.hsps)))
...
gi|301171322|ref|NR_035857.1| 2
gi|262205330|ref|NR_030198.1| 2
gi|301171447|ref|NR_035871.1| 2
gi|262205298|ref|NR_030190.1| 2
gi|270132717|ref|NR_032716.1| 2
```

`hsp_filter` works the same as `hit_filter`, only instead of looking at the `Hit` objects, it performs filtering on the `HSP` objects in each hits.

As for the `map` methods, they too accept a callback function as their arguments. However, instead of returning `True` or `False`, the callback function must return the modified `Hit` or `HSP` object (depending on whether you're using `hit_map` or `hsp_map`).

Let's see an example where we're using `hit_map` to rename the hit IDs:

```
>>> def map_func(hit):
...     # renames "gi|301171322|ref|NR_035857.1|" to "NR_035857.1"
...     hit.id = hit.id.split("|")[3]
...     return hit
...
>>> mapped_qresult = blast_qresult.hit_map(map_func)
>>> for hit in mapped_qresult[:5]:
...     print(hit.id)
...
NR_030195.1
NR_035856.1
NR_032573.1
NR_035857.1
NR_035851.1
```

Again, `hsp_map` works the same as `hit_map`, but on `HSP` objects instead of `Hit` objects.

11.1.2 Hit

`Hit` objects represent all query results from a single database entry. They are the second-level container in the `Bio.SearchIO` object hierarchy. You've seen that they are contained by `QueryResult` objects, but they themselves contain `HSP` objects.

Let's see what they look like, beginning with our BLAST search:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hit = blast_qresult[3]  # fourth hit from the query result
>>> print(blast_hit)
Query: 42291
```

```
      mystery_seq
  Hit: gi|301171322|ref|NR_035857.1| (86)
       Pan troglodytes microRNA mir-520c (MIR520C), microRNA
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
          #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
          0   8.9e-20     100.47      60           [1:61]                 [13:73]
          1   3.3e-06      55.39      60           [0:60]                 [13:73]
```

You see that we've got the essentials covered here:

- The query ID and description is present. A hit is always tied to a query, so we want to keep track of the originating query as well. These values can be accessed from a hit using the `query_id` and `query_description` attributes.
- We also have the unique hit ID, description, and full sequence lengths. They can be accessed using `id`, `description`, and `seq_len`, respectively.
- Finally, there's a table containing quick information about the HSPs this hit contains. In each row, we've got the important HSP details listed: the HSP index, its e-value, its bit score, its span (the alignment length including gaps), its query coordinates, and its hit coordinates.

Now let's contrast this with the BLAT search. Remember that in the BLAT search we had one hit with 17 HSPs.

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_hit = blat_qresult[0]  # the only hit
>>> print(blat_hit)
Query: mystery_seq
       <unknown description>
  Hit: chr19 (59128983)
       <unknown description>
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
          #   E-value  Bit score    Span      Query range              Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
          0         ?          ?       ?           [0:61]    [54204480:54204541]
          1         ?          ?       ?           [0:61]    [54233104:54264463]
          2         ?          ?       ?           [0:61]    [54254477:54260071]
          3         ?          ?       ?           [1:61]    [54210720:54210780]
          4         ?          ?       ?           [0:60]    [54198476:54198536]
          5         ?          ?       ?           [0:61]    [54265610:54265671]
          6         ?          ?       ?           [0:61]    [54238143:54240175]
          7         ?          ?       ?           [0:60]    [54189735:54189795]
          8         ?          ?       ?           [0:61]    [54185425:54185486]
          9         ?          ?       ?           [0:60]    [54197657:54197717]
         10         ?          ?       ?           [0:61]    [54255662:54255723]
         11         ?          ?       ?           [0:61]    [54201651:54201712]
         12         ?          ?       ?           [8:60]    [54206009:54206061]
         13         ?          ?       ?          [10:61]    [54178987:54179038]
         14         ?          ?       ?           [8:61]    [54212018:54212071]
         15         ?          ?       ?           [8:51]    [54234278:54234321]
         16         ?          ?       ?           [8:61]    [54238143:54238196]
```

Here, we've got a similar level of detail as with the BLAST hit we saw earlier. There are some differences worth explaining, though:

- The e-value and bit score column values. As BLAT HSPs do not have e-values and bit scores, the display defaults to '?'.
- What about the span column? The span values is meant to display the complete alignment length, which consists of all residues and any gaps that may be present. The PSL format do not have this information readily available and `Bio.SearchIO` does not attempt to try guess what it is, so we get a '?' similar to the e-value and bit score columns.

In terms of Python objects, `Hit` behaves almost the same as Python lists, but contain `HSP` objects exclusively. If you're familiar with lists, you should encounter no difficulties working with the `Hit` object.

Just like Python lists, `Hit` objects are iterable, and each iteration returns one `HSP` object it contains:

```
>>> for hsp in blast_hit:
...     hsp
...
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
HSP(hit_id='gi|301171322|ref|NR_035857.1|', query_id='42291', 1 fragments)
```

You can invoke `len` on a `Hit` to see how many `HSP` objects it has:

```
>>> len(blast_hit)
2
>>> len(blat_hit)
17
```

You can use the slice notation on `Hit` objects, whether to retrieve single `HSP` or multiple `HSP` objects. Like `QueryResult`, if you slice for multiple `HSP`, a new `Hit` object will be returned containing only the sliced `HSP` objects:

```
>>> blat_hit[0]  # retrieve single items
HSP(hit_id='chr19', query_id='mystery_seq', 1 fragments)
>>> sliced_hit = blat_hit[4:9]  # retrieve multiple items
>>> len(sliced_hit)
5
>>> print(sliced_hit)
Query: mystery_seq
       <unknown description>
  Hit: chr19 (59128983)
       <unknown description>
 HSPs: ----  --------  ---------  ------  ---------------  ----------------------
          #   E-value  Bit score    Span      Query range                Hit range
       ----  --------  ---------  ------  ---------------  ----------------------
          0         ?          ?       ?           [0:60]    [54198476:54198536]
          1         ?          ?       ?           [0:61]    [54265610:54265671]
          2         ?          ?       ?           [0:61]    [54238143:54240175]
          3         ?          ?       ?           [0:60]    [54189735:54189795]
          4         ?          ?       ?           [0:61]    [54185425:54185486]
```

You can also sort the `HSP` inside a `Hit`, using the exact same arguments like the sort method you saw in the `QueryResult` object.

Finally, there are also the `filter` and `map` methods you can use on `Hit` objects. Unlike in the `QueryResult` object, `Hit` objects only have one variant of `filter` (`Hit.filter`) and one variant of `map` (`Hit.map`). Both of `Hit.filter` and `Hit.map` work on the `HSP` objects a `Hit` has.

11.1.3 HSP

`HSP` (high-scoring pair) represents region(s) in the hit sequence that contains significant alignment(s) to the query sequence. It contains the actual match between your query sequence and a database entry. As this match is determined by the sequence search tool's algorithms, the `HSP` object contains the bulk of the statistics computed by the search tool. This also makes the distinction between `HSP` objects from different search tools more apparent compared to the differences you've seen in `QueryResult` or `Hit` objects.

Let's see some examples from our BLAST and BLAT searches. We'll look at the BLAST HSP first:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_hsp = blast_qresult[0][0]  # first hit, first hsp
>>> print(blast_hsp)
      Query: 42291 mystery_seq
        Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
Query range: [0:61] (1)
  Hit range: [0:61] (1)
Quick stats: evalue 4.9e-23; bitscore 111.29
  Fragments: 1 (61 columns)
     Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
             |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
       Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

Just like `QueryResult` and `Hit`, invoking `print` on an `HSP` shows its general details:

- There are the query and hit IDs and descriptions. We need these to identify our `HSP`.
- We've also got the matching range of the query and hit sequences. The slice notation we're using here is an indication that the range is displayed using Python's indexing style (zero-based, half open). The number inside the parenthesis denotes the strand. In this case, both sequences have the plus strand.
- Some quick statistics are available: the e-value and bitscore.
- There is information about the HSP fragments. Ignore this for now; it will be explained later on.
- And finally, we have the query and hit sequence alignment itself.

These details can be accessed on their own using the dot notation, just like in `QueryResult` and `Hit`:

```
>>> blast_hsp.query_range
(0, 61)

>>> blast_hsp.evalue
4.91307e-23
```

They're not the only attributes available, though. `HSP` objects come with a default set of properties that makes it easy to probe their various details. Here are some examples:

```
>>> blast_hsp.hit_start  # start coordinate of the hit sequence
0
>>> blast_hsp.query_span  # how many residues in the query sequence
61
>>> blast_hsp.aln_span  # how long the alignment is
61
```

Check out the `HSP` documentation for a full list of these predefined properties.

Furthermore, each sequence search tool usually computes its own statistics / details for its `HSP` objects. For example, an XML BLAST search also outputs the number of gaps and identical residues. These attributes can be accessed like so:

```
>>> blast_hsp.gap_num  # number of gaps
0
>>> blast_hsp.ident_num  # number of identical residues
61
```

These details are format-specific; they may not be present in other formats. To see which details are available for a given sequence search tool, you should check the format's documentation in `Bio.SearchIO`. Alternatively, you may also use `.__dict__.keys()` for a quick list of what's available:

```
>>> blast_hsp.__dict__.keys()
['bitscore', 'evalue', 'ident_num', 'gap_num', 'bitscore_raw', 'pos_num', '_items']
```

Finally, you may have noticed that the `query` and `hit` attributes of our HSP are not just regular strings:

```
>>> blast_hsp.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id='42291', name='ali
>>> blast_hsp.hit
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id='gi|262205317|ref
```

They are `SeqRecord` objects you saw earlier in Section 4! This means that you can do all sorts of interesting things you can do with `SeqRecord` objects on `HSP.query` and/or `HSP.hit`.

It should not surprise you now that the `HSP` object has an `alignment` property which is a `MultipleSeqAlignment` object:

```
>>> print(blast_hsp.aln)
Alignment with 2 rows and 61 columns
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAG...GGG 42291
CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAG...GGG gi|262205317|ref|NR_030195.1|
```

Having probed the BLAST HSP, let's now take a look at HSPs from our BLAT results for a different kind of HSP. As usual, we'll begin by invoking `print` on it:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_hsp = blat_qresult[0][0]  # first hit, first hsp
>>> print(blat_hsp)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
Quick stats: evalue ?; bitscore ?
  Fragments: 1 (? columns)
```

Some of the outputs you may have already guessed. We have the query and hit IDs and descriptions and the sequence coordinates. Values for evalue and bitscore is '?' as BLAT HSPs do not have these attributes. But The biggest difference here is that you don't see any sequence alignments displayed. If you look closer, PSL formats themselves do not have any hit or query sequences, so `Bio.SearchIO` won't create any sequence or alignment objects. What happens if you try to access `HSP.query`, `HSP.hit`, or `HSP.aln`? You'll get the default values for these attributes, which is `None`:

```
>>> blat_hsp.hit is None
True
>>> blat_hsp.query is None
True
>>> blat_hsp.aln is None
True
```

This does not affect other attributes, though. For example, you can still access the length of the query or hit alignment. Despite not displaying any attributes, the PSL format still have this information so `Bio.SearchIO` can extract them:

```
>>> blat_hsp.query_span  # length of query match
61
>>> blat_hsp.hit_span  # length of hit match
61
```

Other format-specific attributes are still present as well:

```
>>> blat_hsp.score  # PSL score
61
>>> blat_hsp.mismatch_num  # the mismatch column
0
```

So far so good? Things get more interesting when you look at another 'variant' of HSP present in our BLAT results. You might recall that in BLAT searches, sometimes we get our results separated into 'blocks'. These blocks are essentially alignment fragments that may have some intervening sequence between them.

Let's take a look at a BLAT HSP that contains multiple blocks to see how `Bio.SearchIO` deals with this:

```
>>> blat_hsp2 = blat_qresult[0][1]  # first hit, second hsp
>>> print(blat_hsp2)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54233104:54264463] (1)
Quick stats: evalue ?; bitscore ?
  Fragments: ---  --------------  ----------------------  ----------------------
               #            Span             Query range               Hit range
             ---  --------------  ----------------------  ----------------------
               0               ?                 [0:18]     [54233104:54233122]
               1               ?                [18:61]     [54264420:54264463]
```

What's happening here? We still some essential details covered: the IDs and descriptions, the coordinates, and the quick statistics are similar to what you've seen before. But the fragments detail is all different. Instead of showing 'Fragments: 1', we now have a table with two data rows.

This is how `Bio.SearchIO` deals with HSPs having multiple fragments. As mentioned before, an HSP alignment may be separated by intervening sequences into fragments. The intervening sequences are not part of the query-hit match, so they should not be considered part of query nor hit sequence. However, they do affect how we deal with sequence coordinates, so we can't ignore them.

Take a look at the hit coordinate of the HSP above. In the `Hit range:` field, we see that the coordinate is `[54233104:54264463]`. But looking at the table rows, we see that not the entire region spanned by this coordinate matches our query. Specifically, the intervening region spans from `54233122` to `54264420`.

Why then, is the query coordinates seem to be contiguous, you ask? This is perfectly fine. In this case it means that the query match is contiguous (no intervening regions), while the hit match is not.

All these attributes are accessible from the HSP directly, by the way:

```
>>> blat_hsp2.hit_range  # hit start and end coordinates of the entire HSP
(54233104, 54264463)
>>> blat_hsp2.hit_range_all  # hit start and end coordinates of each fragment
[(54233104, 54233122), (54264420, 54264463)]
>>> blat_hsp2.hit_span  # hit span of the entire HSP
31359
>>> blat_hsp2.hit_span_all  # hit span of each fragment
[18, 43]
>>> blat_hsp2.hit_inter_ranges  # start and end coordinates of intervening regions in the hit sequence
[(54233122, 54264420)]
>>> blat_hsp2.hit_inter_spans  # span of intervening regions in the hit sequence
[31298]
```

Most of these attributes are not readily available from the PSL file we have, but `Bio.SearchIO` calculates them for you on the fly when you parse the PSL file. All it needs are the start and end coordinates of each fragment.

What about the `query`, `hit`, and `aln` attributes? If the HSP has multiple fragments, you won't be able to use these attributes as they only fetch single `SeqRecord` or `MultipleSeqAlignment` objects. However, you can use their `*_all` counterparts: `query_all`, `hit_all`, and `aln_all`. These properties will return a list containing `SeqRecord` or `MultipleSeqAlignment` objects from each of the HSP fragment. There are other attributes that behave similarly, i.e. they only work for HSPs with one fragment. Check out the `HSP` documentation for a full list.

Finally, to check whether you have multiple fragments or not, you can use the `is_fragmented` property like so:

```
>>> blat_hsp2.is_fragmented  # BLAT HSP with 2 fragments
True
>>> blat_hsp.is_fragmented  # BLAT HSP from earlier, with one fragment
False
```

Before we move on, you should also know that we can use the slice notation on `HSP` objects, just like `QueryResult` or `Hit` objects. When you use this notation, you'll get an `HSPFragment` object in return, the last component of the object model.

11.1.4 HSPFragment

`HSPFragment` represents a single, contiguous match between the query and hit sequences. You could consider it the core of the object model and search result, since it is the presence of these fragments that determine whether your search have results or not.

In most cases, you don't have to deal with `HSPFragment` objects directly since not that many sequence search tools fragment their HSPs. When you do have to deal with them, what you should remember is that `HSPFragment` objects were written with to be as compact as possible. In most cases, they only contain attributes directly related to sequences: strands, reading frames, molecule types, coordinates, the sequences themselves, and their IDs and descriptions.

These attributes are readily shown when you invoke `print` on an `HSPFragment`. Here's an example, taken from our BLAST search:

```
>>> from Bio import SearchIO
>>> blast_qresult = SearchIO.read("my_blast.xml", "blast-xml")
>>> blast_frag = blast_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blast_frag)
      Query: 42291 mystery_seq
        Hit: gi|262205317|ref|NR_030195.1| Homo sapiens microRNA 520b (MIR520...
```

```
Query range: [0:61] (1)
  Hit range: [0:61] (1)
  Fragments: 1 (61 columns)
     Query - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
       Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

At this level, the BLAT fragment looks quite similar to the BLAST fragment, save for the query and hit sequences which are not present:

```
>>> blat_qresult = SearchIO.read("my_blat.psl", "blat-psl")
>>> blat_frag = blat_qresult[0][0][0]  # first hit, first hsp, first fragment
>>> print(blat_frag)
      Query: mystery_seq <unknown description>
        Hit: chr19 <unknown description>
Query range: [0:61] (1)
  Hit range: [54204480:54204541] (1)
  Fragments: 1 (? columns)
```

In all cases, these attributes are accessible using our favorite dot notation. Some examples:

```
>>> blast_frag.query_start  # query start coordinate
0
>>> blast_frag.hit_strand  # hit sequence strand
1
>>> blast_frag.hit  # hit sequence, as a SeqRecord object
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG'), id='gi|262205317|ref
```

11.2 A note about standards and conventions

Before we move on to the main functions, there is something you ought to know about the standards `Bio.SearchIO` uses. If you've worked with multiple sequence search tools, you might have had to deal with the many different ways each program deals with things like sequence coordinates. It might not have been a pleasant experience as these search tools usually have their own standards. For example, one tools might use one-based coordinates, while the other uses zero-based coordinates. Or, one program might reverse the start and end coordinates if the strand is minus, while others don't. In short, these often creates unnecessary mess must be dealt with.

We realize this problem ourselves and we intend to address it in `Bio.SearchIO`. After all, one of the goals of `Bio.SearchIO` is to create a common, easy to use interface to deal with various search output files. This means creating standards that extend beyond the object model you just saw.

Now, you might complain, "Not another standard!". Well, eventually we have to choose one convention or the other, so this is necessary. Plus, we're not creating something entirely new here; just adopting a standard we think is best for a Python programmer (it is Biopython, after all).

There are three implicit standards that you can expect when working with `Bio.SearchIO`:

- The first one pertains to sequence coordinates. In `Bio.SearchIO`, all sequence coordinates follows Python's coordinate style: zero-based and half open. For example, if in a BLAST XML output file the start and end coordinates of an HSP are 10 and 28, they would become 9 and 28 in `Bio.SearchIO`. The start coordinate becomes 9 because Python indices start from zero, while the end coordinate remains 28 as Python slices omit the last item in an interval.
- The second is on sequence coordinate orders. In `Bio.SearchIO`, start coordinates are always less than or equal to end coordinates. This isn't always the case with all sequence search tools, as some of them have larger start coordinates when the sequence strand is minus.

- The last one is on strand and reading frame values. For strands, there are only four valid choices: `1` (plus strand), `-1` (minus strand), `0` (protein sequences), and `None` (no strand). For reading frames, the valid choices are integers from `-3` to `3` and `None`.

Note that these standards only exist in `Bio.SearchIO` objects. If you write `Bio.SearchIO` objects into an output format, `Bio.SearchIO` will use the format's standard for the output. It does not force its standard over to your output file.

11.3 Reading search output files

There are two functions you can use for reading search output files into `Bio.SearchIO` objects: `read` and `parse`. They're essentially similar to `read` and `parse` functions in other submodules like `Bio.SeqIO` or `Bio.AlignIO`. In both cases, you need to supply the search output file name and the file format name, both as Python strings. You can check the documentation for a list of format names `Bio.SearchIO` recognizes.

`Bio.SearchIO.read` is used for reading search output files with only one query and returns a `QueryResult` object. You've seen `read` used in our previous examples. What you haven't seen is that `read` may also accept additional keyword arguments, depending on the file format.

Here are some examples. In the first one, we use `read` just like previously to read a BLAST tabular output file. In the second one, we use a keyword argument to modify so it parses the BLAST tabular variant with comments in it:

```
>>> from Bio import SearchIO
>>> qresult = SearchIO.read("tab_2226_tblastn_003.txt", "blast-tab")
>>> qresult
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> qresult2 = SearchIO.read("tab_2226_tblastn_007.txt", "blast-tab", comments=True)
>>> qresult2
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
```

These keyword arguments differs among file formats. Check the format documentation to see if it has keyword arguments that modifies its parser's behavior.

As for the `Bio.SearchIO.parse`, it is used for reading search output files with any number of queries. The function returns a generator object that yields a `QueryResult` object in each iteration. Like `Bio.SearchIO.read`, it also accepts format-specific keyword arguments:

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("tab_2226_tblastn_001.txt", "blast-tab")
>>> for qresult in qresults:
...     print(qresult.id)
...
gi|16080617|ref|NP_391444.1|
gi|11464971:4-101
>>> qresults2 = SearchIO.parse("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> for qresult in qresults2:
...     print(qresult.id)
...
random_s00
gi|16080617|ref|NP_391444.1|
gi|11464971:4-101
```

11.4 Dealing with large search output files with indexing

Sometimes, you're handed a search output file containing hundreds or thousands of queries that you need to parse. You can of course use `Bio.SearchIO.parse` for this file, but that would be grossly inefficient if you need to access only a few of the queries. This is because `parse` will parse all queries it sees before it fetches your query of interest.

In this case, the ideal choice would be to index the file using `Bio.SearchIO.index` or `Bio.SearchIO.index_db`. If the names sound familiar, it's because you've seen them before in Section 5.4.2. These functions also behave similarly to their `Bio.SeqIO` counterparts, with the addition of format-specific keyword arguments.

Here are some examples. You can use `index` with just the filename and format name:

```
>>> from Bio import SearchIO
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab")
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|']
>>> idx["gi|16080617|ref|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or also with the format-specific keyword argument:

```
>>> idx = SearchIO.index("tab_2226_tblastn_005.txt", "blast-tab", comments=True)
>>> sorted(idx.keys())
['gi|11464971:4-101', 'gi|16080617|ref|NP_391444.1|', 'random_s00']
>>> idx["gi|16080617|ref|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

Or with the `key_function` argument, as in `Bio.SeqIO`:

```
>>> key_function = lambda id: id.upper()  # capitalizes the keys
>>> idx = SearchIO.index("tab_2226_tblastn_001.txt", "blast-tab", key_function=key_function)
>>> sorted(idx.keys())
['GI|11464971:4-101', 'GI|16080617|REF|NP_391444.1|']
>>> idx["GI|16080617|REF|NP_391444.1|"]
QueryResult(id='gi|16080617|ref|NP_391444.1|', 3 hits)
>>> idx.close()
```

`Bio.SearchIO.index_db` works like as `index`, only it writes the query offsets into an SQLite database file.

11.5 Writing and converting search output files

It is occasionally useful to be able to manipulate search results from an output file and write it again to a new file. `Bio.SearchIO` provides a `write` function that lets you do exactly this. It takes as its arguments an iterable returning `QueryResult` objects, the output filename to write to, the format name to write to, and optionally some format-specific keyword arguments. It returns a four-item tuple, which denotes the number or `QueryResult`, `Hit`, `HSP`, and `HSPFragment` objects that were written.

```
>>> from Bio import SearchIO
>>> qresults = SearchIO.parse("mirna.xml", "blast-xml")  # read XML file
>>> SearchIO.write(qresults, "results.tab", "blast-tab")  # write to tabular file
(3, 239, 277, 277)
```

You should note different file formats require different attributes of the `QueryResult`, `Hit`, `HSP` and `HSPFragment` objects. If these attributes are not present, writing won't work. In other words, you can't always write to the output format that you want. For example, if you read a BLAST XML file, you wouldn't be able to write the results to a PSL file as PSL files require attributes not calculated by BLAST (e.g. the number of repeat matches). You can always set these attributes manually, if you really want to write to PSL, though.

Like `read`, `parse`, `index`, and `index_db`, `write` also accepts format-specific keyword arguments. Check out the documentation for a complete list of formats `Bio.SearchIO` can write to and their arguments.

Finally, `Bio.SearchIO` also provides a `convert` function, which is simply a shortcut for `Bio.SearchIO.parse` and `Bio.SearchIO.write`. Using the convert function, our example above would be:

```
>>> from Bio import SearchIO
>>> SearchIO.convert("mirna.xml", "blast-xml", "results.tab", "blast-tab")
(3, 239, 277, 277)
```

As `convert` uses `write`, it is only limited to format conversions that have all the required attributes. Here, the BLAST XML file provides all the default values a BLAST tabular file requires, so it works just fine. However, other format conversions are less likely to work since you need to manually assign the required attributes first.

Chapter 12

Accessing NCBI's Entrez databases

Entrez (https://www.ncbi.nlm.nih.gov/Web/Search/entrezfs.html) is a data retrieval system that provides users access to NCBI's databases such as PubMed, GenBank, GEO, and many others. You can access Entrez from a web browser to manually enter queries, or you can use Biopython's `Bio.Entrez` module for programmatic access to Entrez. The latter allows you for example to search PubMed or download GenBank records from within a Python script.

The `Bio.Entrez` module makes use of the Entrez Programming Utilities (also known as EUtils), consisting of eight tools that are described in detail on NCBI's page at https://www.ncbi.nlm.nih.gov/books/NBK25501/. Each of these tools corresponds to one Python function in the `Bio.Entrez` module, as described in the sections below. This module makes sure that the correct URL is used for the queries, and that NCBI's guidelines for responsible data access are being followed.

The output returned by the Entrez Programming Utilities is typically in XML format. To parse such output, you have several options:

1. Use `Bio.Entrez`'s parser to parse the XML output into a Python object;
2. Use one of the XML parsers available in Python's standard library;
3. Read the XML output as raw text, and parse it by string searching and manipulation.

See the Python documentation for a description of the XML parsers in Python's standard library. Here, we discuss the parser in Biopython's `Bio.Entrez` module. This parser can be used to parse data as provided through `Bio.Entrez`'s programmatic access functions to Entrez, but can also be used to parse XML data from NCBI Entrez that are stored in a file. In the latter case, the XML file should be opened in binary mode (e.g. `open("myfile.xml", "rb")`) for the XML parser in `Bio.Entrez` to work correctly. Alternatively, you can pass the file name or path to the XML file, and let `Bio.Entrez` take care of opening and closing the file.

NCBI uses DTD (Document Type Definition) files to describe the structure of the information contained in XML files. Most of the DTD files used by NCBI are included in the Biopython distribution. The `Bio.Entrez` parser makes use of the DTD files when parsing an XML file returned by NCBI Entrez.

Occasionally, you may find that the DTD file associated with a specific XML file is missing in the Biopython distribution. In particular, this may happen when NCBI updates its DTD files. If this happens, `Entrez.read` will show a warning message with the name and URL of the missing DTD file. The parser will proceed to access the missing DTD file through the internet, allowing the parsing of the XML file to continue. However, the parser is much faster if the DTD file is available locally. For this purpose, please download the DTD file from the URL in the warning message and place it in the directory `...site-packages/Bio/Entrez/DTDs`, containing the other DTD files. If you don't have write access to this directory, you can also place the DTD file in `~/.biopython/Bio/Entrez/DTDs`, where `~` represents your home directory. Since this directory is read before the directory `...site-packages/Bio/Entrez/DTDs`, you can also put newer versions of DTD files there if the ones in `...site-packages/Bio/Entrez/DTDs` become

outdated. Alternatively, if you installed Biopython from source, you can add the DTD file to the source code's `Bio/Entrez/DTDs` directory, and reinstall Biopython. This will install the new DTD file in the correct location together with the other DTD files.

The Entrez Programming Utilities can also generate output in other formats, such as the Fasta or GenBank file formats for sequence databases, or the MedLine format for the literature database, discussed in Section 12.13.

The functions in `Bio.Entrez` for programmatic access to Entrez return data either in binary format or in text format, depending on the type of data requested. In most cases, these functions return data in text format by decoding the data obtained from NCBI Entrez to Python strings under the assumption that the encoding is UTF-8. However, XML data are returned in binary format. The reason for this is that the encoding is specified in the XML document itself, which means that we won't know the correct encoding to use until we start parsing the file. `Bio.Entrez`'s parser therefore accepts data in binary format, extracts the encoding from the XML, and uses it to decode all text in the XML document to Python strings, ensuring that all text (in particular in languages other than English) are interpreted correctly. This is also the reason why you should open an XML file a binary mode when you want to use `Bio.Entrez`'s parser to parse the file.

12.1 Entrez Guidelines

Before using Biopython to access the NCBI's online resources (via `Bio.Entrez` or some of the other modules), please read the NCBI's Entrez User Requirements. If the NCBI finds you are abusing their systems, they can and will ban your access!

To paraphrase:

- For any series of more than 100 requests, do this at weekends or outside USA peak times. This is up to you to obey.
- Use the `https://eutils.ncbi.nlm.nih.gov` address, not the standard NCBI Web address. Biopython uses this web address.
- If you are using a API key, you can make at most 10 queries per second, otherwise at most 3 queries per second. This is automatically enforced by Biopython. Include `api_key="MyAPIkey"` in the argument list or set it as a module level variable:

  ```
  >>> from Bio import Entrez
  >>> Entrez.api_key = "MyAPIkey"
  ```

- Use the optional email parameter so the NCBI can contact you if there is a problem. You can either explicitly set this as a parameter with each call to Entrez (e.g. include `email="A.N.Other@example.com"` in the argument list), or you can set a global email address:

  ```
  >>> from Bio import Entrez
  >>> Entrez.email = "A.N.Other@example.com"
  ```

  `Bio.Entrez` will then use this email address with each call to Entrez. The `example.com` address is a reserved domain name specifically for documentation (RFC 2606). Please DO NOT use a random email – it's better not to give an email at all. The email parameter has been mandatory since June 1, 2010. In case of excessive usage, NCBI will attempt to contact a user at the e-mail address provided prior to blocking access to the E-utilities.

- If you are using Biopython within some larger software suite, use the tool parameter to specify this. You can either explicitly set the tool name as a parameter with each call to Entrez (e.g. include `tool="MyLocalScript"` in the argument list), or you can set a global tool name:

```
>>> from Bio import Entrez
>>> Entrez.tool = "MyLocalScript"
```

The tool parameter will default to Biopython.

- For large queries, the NCBI also recommend using their session history feature (the WebEnv session cookie string, see Section 12.16). This is only slightly more complicated.

In conclusion, be sensible with your usage levels. If you plan to download lots of data, consider other options. For example, if you want easy access to all the human genes, consider fetching each chromosome by FTP as a GenBank file, and importing these into your own BioSQL database (see Section 22.3).

12.2 EInfo: Obtaining information about the Entrez databases

EInfo provides field index term counts, last update, and available links for each of NCBI's databases. In addition, you can use EInfo to obtain a list of all database names accessible through the Entrez utilities:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo()
>>> result = stream.read()
>>> stream.close()
```

The variable `result` now contains a list of databases in XML format:

```
>>> print(result)
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN"
 "https://www.ncbi.nlm.nih.gov/entrez/query/DTD/eInfo_020511.dtd">
<eInfoResult>
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nucleotide</DbName>
        <DbName>nuccore</DbName>
        <DbName>nucgss</DbName>
        <DbName>nucest</DbName>
        <DbName>structure</DbName>
        <DbName>genome</DbName>
        <DbName>books</DbName>
        <DbName>cancerchromosomes</DbName>
        <DbName>cdd</DbName>
        <DbName>gap</DbName>
        <DbName>domains</DbName>
        <DbName>gene</DbName>
        <DbName>genomeprj</DbName>
        <DbName>gensat</DbName>
        <DbName>geo</DbName>
        <DbName>gds</DbName>
        <DbName>homologene</DbName>
        <DbName>journals</DbName>
        <DbName>mesh</DbName>
        <DbName>ncbisearch</DbName>
```

```
        <DbName>nlmcatalog</DbName>
        <DbName>omia</DbName>
        <DbName>omim</DbName>
        <DbName>pmc</DbName>
        <DbName>popset</DbName>
        <DbName>probe</DbName>
        <DbName>proteinclusters</DbName>
        <DbName>pcassay</DbName>
        <DbName>pccompound</DbName>
        <DbName>pcsubstance</DbName>
        <DbName>snp</DbName>
        <DbName>taxonomy</DbName>
        <DbName>toolkit</DbName>
        <DbName>unigene</DbName>
        <DbName>unists</DbName>
</DbList>
</eInfoResult>
```

Since this is a fairly simple XML file, we could extract the information it contains simply by string searching. Using `Bio.Entrez`'s parser instead, we can directly parse this XML file into a Python object:

```
>>> from Bio import Entrez
>>> stream = Entrez.einfo()
>>> record = Entrez.read(stream)
```

Now `record` is a dictionary with exactly one key:

```
>>> record.keys()
dict_keys(['DbList'])
```

The values stored in this key is the list of database names shown in the XML above:

```
>>> record["DbList"]
['pubmed', 'protein', 'nucleotide', 'nuccore', 'nucgss', 'nucest',
 'structure', 'genome', 'books', 'cancerchromosomes', 'cdd', 'gap',
 'domains', 'gene', 'genomeprj', 'gensat', 'geo', 'gds', 'homologene',
 'journals', 'mesh', 'ncbisearch', 'nlmcatalog', 'omia', 'omim', 'pmc',
 'popset', 'probe', 'proteinclusters', 'pcassay', 'pccompound',
 'pcsubstance', 'snp', 'taxonomy', 'toolkit', 'unigene', 'unists']
```

For each of these databases, we can use EInfo again to obtain more information:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.einfo(db="pubmed")
>>> record = Entrez.read(stream)
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'

>>> record["DbInfo"]["Count"]
'17989604'
>>> record["DbInfo"]["LastUpdate"]
'2008/05/24 06:45'
```

Try `record["DbInfo"].keys()` for other information stored in this record. One of the most useful is a list of possible search fields for use with ESearch:

```
>>> for field in record["DbInfo"]["FieldList"]:
...     print("%(Name)s, %(FullName)s, %(Description)s" % field)
...
ALL, All Fields, All terms from all searchable fields
UID, UID, Unique number assigned to publication
FILT, Filter, Limits the records
TITL, Title, Words in title of publication
WORD, Text Word, Free text associated with publication
MESH, MeSH Terms, Medical Subject Headings assigned to publication
MAJR, MeSH Major Topic, MeSH terms of major importance to publication
AUTH, Author, Author(s) of publication
JOUR, Journal, Journal abbreviation of publication
AFFL, Affiliation, Author's institutional affiliation and address
...
```

That's a long list, but indirectly this tells you that for the PubMed database, you can do things like `Jones[AUTH]` to search the author field, or `Sanger[AFFL]` to restrict to authors at the Sanger Centre. This can be very handy - especially if you are not so familiar with a particular database.

12.3 ESearch: Searching the Entrez databases

To search any of these databases, we use `Bio.Entrez.esearch()`. For example, let's search in PubMed for publications that include Biopython in their title:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="biopython[title]", retmax="40")
>>> record = Entrez.read(stream)
>>> "19304878" in record["IdList"]
True

>>> print(record["IdList"])
['22909249', '19304878']
```

In this output, you see PubMed IDs (including 19304878 which is the PMID for the Biopython application note), which can be retrieved by EFetch (see section 12.6).

You can also use ESearch to search GenBank. Here we'll do a quick search for the *matK* gene in *Cypripedioideae* orchids (see Section 12.2 about EInfo for one way to find out which fields you can search in each Entrez database):

```
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Cypripedioideae[Orgn] AND matK[Gene]", idtype="acc"
... )
>>> record = Entrez.read(stream)
>>> record["Count"]
'348'
>>> record["IdList"]
['JQ660909.1', 'JQ660908.1', 'JQ660907.1', 'JQ660906.1', ..., 'JQ660890.1']
```

Each of the IDs (JQ660909.1, JQ660908.1, JQ660907.1, ...) is a GenBank identifier (Accession number). See section 12.6 for information on how to actually download these GenBank records.

Note that instead of a species name like `Cypripedioideae[Orgn]`, you can restrict the search using an NCBI taxon identifier, here this would be `txid158330[Orgn]`. This isn't currently documented on the

ESearch help page - the NCBI explained this in reply to an email query. You can often deduce the search term formatting by playing with the Entrez web interface. For example, including `complete[prop]` in a genome search restricts to just completed genomes.

As a final example, let's get a list of computational journal titles:

```
>>> stream = Entrez.esearch(db="nlmcatalog", term="computational[Journal]", retmax="20")
>>> record = Entrez.read(stream)
>>> print("{} computational journals found".format(record["Count"]))
117 computational Journals found
>>> print("The first 20 are\n{}".format(record["IdList"]))
['101660833', '101664671', '101661657', '101659814', '101657941',
 '101653734', '101669877', '101649614', '101647835', '101639023',
 '101627224', '101647801', '101589678', '101585369', '101645372',
 '101586429', '101582229', '101574747', '101564639', '101671907']
```

Again, we could use EFetch to obtain more information for each of these journal IDs.

ESearch has many useful options — see the ESearch help page for more information.

12.4 EPost: Uploading a list of identifiers

EPost uploads a list of UIs for use in subsequent search strategies; see the EPost help page for more information. It is available from Biopython through the `Bio.Entrez.epost()` function.

To give an example of when this is useful, suppose you have a long list of IDs you want to download using EFetch (maybe sequences, maybe citations – anything). When you make a request with EFetch your list of IDs, the database etc, are all turned into a long URL sent to the server. If your list of IDs is long, this URL gets long, and long URLs can break (e.g. some proxies don't cope well).

Instead, you can break this up into two steps, first uploading the list of IDs using EPost (this uses an "HTML post" internally, rather than an "HTML get", getting round the long URL problem). With the history support, you can then refer to this long list of IDs, and download the associated data with EFetch.

Let's look at a simple example to see how EPost works – uploading some PubMed identifiers:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> id_list = ["19304878", "18606172", "16403221", "16377612", "14871861", "14630660"]
>>> print(Entrez.epost("pubmed", id=",".join(id_list)).read())
<?xml version="1.0"?>
<!DOCTYPE ePostResult PUBLIC "-//NLM//DTD ePostResult, 11 May 2002//EN"
 "https://www.ncbi.nlm.nih.gov/entrez/query/DTD/ePost_020511.dtd">
<ePostResult>
        <QueryKey>1</QueryKey>
        <WebEnv>NCID_01_206841095_130.14.22.101_9001_1242061629</WebEnv>
</ePostResult>
```

The returned XML includes two important strings, `QueryKey` and `WebEnv` which together define your history session. You would extract these values for use with another Entrez call such as EFetch:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> id_list = ["19304878", "18606172", "16403221", "16377612", "14871861", "14630660"]
>>> search_results = Entrez.read(Entrez.epost("pubmed", id=",".join(id_list)))
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Section 12.16 shows how to use the history feature.

12.5 ESummary: Retrieving summaries from primary IDs

ESummary retrieves document summaries from a list of primary IDs (see the ESummary help page for more information). In Biopython, ESummary is available as `Bio.Entrez.esummary()`. Using the search result above, we can for example find out more about the journal with ID 30367:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="nlmcatalog", id="101660833")
>>> record = Entrez.read(stream)
>>> info = record[0]["TitleMainList"][0]
>>> print("Journal info\nid: {}\nTitle: {}".format(record[0]["Id"], info["Title"]))
Journal info
id: 101660833
Title: IEEE transactions on computational imaging.
```

12.6 EFetch: Downloading full records from Entrez

EFetch is what you use when you want to retrieve a full record from Entrez. This covers several possible databases, as described on the main EFetch Help page.

For most of their databases, the NCBI support several different file formats. Requesting a specific file format from Entrez using `Bio.Entrez.efetch()` requires specifying the `rettype` and/or `retmode` optional arguments. The different combinations are described for each database type on the pages linked to on NCBI efetch webpage.

One common usage is downloading sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with `Bio.SeqIO`, see Sections 5.3.1 and 12.6). From the *Cypripedioideae* example above, we can download GenBank record EU490707 using `Bio.Entrez.efetch`:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> print(stream.read())
LOCUS       EU490707               1302 bp    DNA     linear   PLN 26-JUL-2016
DEFINITION  Selenipedium aequinoctiale maturase K (matK) gene, partial cds;
            chloroplast.
ACCESSION   EU490707
VERSION     EU490707.1
KEYWORDS    .
SOURCE      chloroplast Selenipedium aequinoctiale
  ORGANISM  Selenipedium aequinoctiale
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
            Spermatophyta; Magnoliopsida; Liliopsida; Asparagales; Orchidaceae;
            Cypripedioideae; Selenipedium.
REFERENCE   1  (bases 1 to 1302)
  AUTHORS   Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A., Endara,L.,
            Williams,N.H. and Moore,M.
  TITLE     Phylogenetic utility of ycf1 in orchids: a plastid gene more
            variable than matK
  JOURNAL   Plant Syst. Evol. 277 (1-2), 75-84 (2009)
REFERENCE   2  (bases 1 to 1302)
  AUTHORS   Neubig,K.M., Whitten,W.M., Carlsward,B.S., Blanco,M.A.,
            Endara,C.L., Williams,N.H. and Moore,M.J.
```

```
  TITLE     Direct Submission
  JOURNAL   Submitted (14-FEB-2008) Department of Botany, University of
            Florida, 220 Bartram Hall, Gainesville, FL 32611-8526, USA
FEATURES             Location/Qualifiers
     source          1..1302
                     /organism="Selenipedium aequinoctiale"
                     /organelle="plastid:chloroplast"
                     /mol_type="genomic DNA"
                     /specimen_voucher="FLAS:Blanco 2475"
                     /db_xref="taxon:256374"
     gene            <1..>1302
                     /gene="matK"
     CDS             <1..>1302
                     /gene="matK"
                     /codon_start=1
                     /transl_table=11
                     /product="maturase K"
                     /protein_id="ACC99456.1"
                     /translation="IFYEPVEIFGYDNKSSLVLVKRLITRMYQQNFLISSVNDSNQKG
                     FWGHKHFFSSHFSSQMVSEGFGVILEIPFSSQLVSSLEEKKIPKYQNLRSIHSIFPFL
                     EDKFLHLNYVSDLLIPHPIHLEILVQILQCRIKDVPSLHLLRLLFHEYHNLNSLITSK
                     KFIYAFSKRKKRFLWLLYNSYVYECEYLFQFLRKQSSYLRSTSSGVFLERTHLYVKIE
                     HLLVVCCNSFQRILCFLKDPFMHYVRYQGKAILASKGTLILMKKWKFHLVNFWQSYFH
                     FWSQPYRIHIKQLSNYSFSFLGYFSSVLENHLVVRNQMLENSFIINLLTKKFDTIAPV
                     ISLIGSLSKAQFCTVLGHPISKPIWTDFSDSDILDRFCRICRNLCRYHSGSSKKQVLY
                     RIKYILRLSCARTLARKHKSTVRTFMRRLGSGLLEEFFMEEE"
ORIGIN
        1 atttttacg aacctgtgga aatttttggt tatgacaata aatctagttt agtacttgtg
       61 aaacgtttaa ttactcgaat gtatcaacag aatttttga tttcttcggt taatgattct
      121 aaccaaaaag gattttgggg gcacaagcat tttttttctt ctcatttttc ttctcaaatg
      181 gtatcagaag gttttggagt cattctggaa attccattct cgtcgcaatt agtatcttct
      241 cttgaagaaa aaaaaatacc aaaatatcag aatttacgat ctattcattc aatatttccc
      301 tttttagaag acaaattttt acatttgaat tatgtgtcag atctactaat accccatccc
      361 atccatctgg aaatcttggt tcaaatcctt caatgccgga tcaaggatgt tccttctttg
      421 catttattgc gattgctttt ccacgaatat cataatttga atagtctcat tacttcaaag
      481 aaattcattt acgccttttc aaaaagaaag aaaagattcc tttggttact atataattct
      541 tatgtatatg aatgcgaata tctattccag tttcttcgta aacagtcttc ttatttacga
      601 tcaacatctt ctggagtctt tcttgagcga acacatttat atgtaaaaat agaacatctt
      661 ctagtagtgt gttgtaattc ttttcagagg atcctatgct ttctcaagga tcctttcatg
      721 cattatgttc gatatcaagg aaaagcaatt ctggcttcaa agggaactct tattctgatg
      781 aagaaatgga aatttcatct tgtgaatttt tggcaatctt attttcactt ttggtctcaa
      841 ccgtatagga ttcatataaa gcaattatcc aactattcct tctcttttct ggggtatttt
      901 tcaagtgtac tagaaaatca tttggtagta agaaatcaaa tgctagagaa ttcatttata
      961 ataaatcttc tgactaagaa attcgatacc atagccccag ttatttctct tattggatca
     1021 ttgtcgaaag ctcaattttg tactgtattg ggtcatccta ttagtaaacc gatctggacc
     1081 gatttctcgg attctgatat tcttgatcga ttttgccgga tatgtagaaa tctttgtcgt
     1141 tatcacagcg gatcctcaaa aaaacaggtt ttgtatcgta taaaatatat acttcgactt
     1201 tcgtgtgcta gaactttggc acggaaacat aaaagtacag tacgcacttt tatgcgaaga
     1261 ttaggttcgg gattattaga agaattcttt atggaagaag aa
//
<BLANKLINE>
```

```
<BLANKLINE>
```

Please be aware that as of October 2016 GI identifiers are discontinued in favor of accession numbers. You can still fetch sequences based on their GI, but new sequences are no longer given this identifier. You should instead refer to them by the "Accession number" as done in the example.

The arguments `rettype="gb"` and `retmode="text"` let us download this record in the GenBank format.

Note that until Easter 2009, the Entrez EFetch API let you use "genbank" as the return type, however the NCBI now insist on using the official return types of "gb" or "gbwithparts" (or "gp" for proteins) as described on online. Also note that until Feb 2012, the Entrez EFetch API would default to returning plain text files, but now defaults to XML.

Alternatively, you could for example use `rettype="fasta"` to get the Fasta-format; see the EFetch Sequences Help page for other options. Remember – the available formats depend on which database you are downloading from - see the main EFetch Help page.

If you fetch the record in one of the formats accepted by `Bio.SeqIO` (see Chapter 5), you could directly parse it into a `SeqRecord`:

```
>>> from Bio import SeqIO
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
>>> record = SeqIO.read(stream, "genbank")
>>> stream.close()
>>> print(record.id)
EU490707.1
>>> print(record.name)
EU490707
>>> print(record.description)
Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast
>>> print(len(record.features))
3
>>> record.seq
Seq('ATTTTTTACGAACCTGTGGAAATTTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA')
```

Note that a more typical use would be to save the sequence data to a local file, and *then* parse it with `Bio.SeqIO`. This can save you having to re-download the same file repeatedly while working on your script, and places less load on the NCBI's servers. For example:

```
import os
from Bio import SeqIO
from Bio import Entrez

Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
filename = "EU490707.gbk"
if not os.path.isfile(filename):
    # Downloading...
    stream = Entrez.efetch(db="nucleotide", id="EU490707", rettype="gb", retmode="text")
    output = open(filename, "w")
    output.write(streame.read())
    output.close()
    stream.close()
    print("Saved")

print("Parsing...")
```

```
record = SeqIO.read(filename, "genbank")
print(record)
```

To get the output in XML format, which you can parse using the `Bio.Entrez.read()` function, use `retmode="xml"`:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.efetch(db="nucleotide", id="EU490707", retmode="xml")
>>> record = Entrez.read(stream)
>>> stream.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'
```

So, that dealt with sequences. For examples of parsing file formats specific to the other databases (e.g. the `MEDLINE` format used in PubMed), see Section 12.13.

If you want to perform a search with `Bio.Entrez.esearch()`, and then download the records with `Bio.Entrez.efetch()`, you should use the WebEnv history feature – see Section 12.16.

12.7 ELink: Searching for related items in NCBI Entrez

ELink, available from Biopython as `Bio.Entrez.elink()`, can be used to find related items in the NCBI Entrez databases. For example, you can us this to find nucleotide entries for an entry in the gene database, and other cool stuff.

Let's use ELink to find articles related to the Biopython application note published in *Bioinformatics* in 2009. The PubMed ID of this article is 19304878:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "19304878"
>>> record = Entrez.read(Entrez.elink(dbfrom="pubmed", id=pmid))
```

The `record` variable consists of a Python list, one for each database in which we searched. Since we specified only one PubMed ID to search for, `record` contains only one item. This item is a dictionary containing information about our search term, as well as all the related items that were found:

```
>>> record[0]["DbFrom"]
'pubmed'
>>> record[0]["IdList"]
['19304878']
```

The `"LinkSetDb"` key contains the search results, stored as a list consisting of one item for each target database. In our search results, we only find hits in the PubMed database (although sub-divided into categories):

```
>>> len(record[0]["LinkSetDb"])
8
```

The exact numbers should increase over time:

```
>>> for linksetdb in record[0]["LinkSetDb"]:
...     print(linksetdb["DbTo"], linksetdb["LinkName"], len(linksetdb["Link"]))
...
pubmed pubmed_pubmed 284
pubmed pubmed_pubmed_alsoviewed 7
pubmed pubmed_pubmed_citedin 926
pubmed pubmed_pubmed_combined 6
pubmed pubmed_pubmed_five 6
pubmed pubmed_pubmed_refs 17
pubmed pubmed_pubmed_reviews 12
pubmed pubmed_pubmed_reviews_five 6
```

The actual search results are stored as under the `"Link"` key.

Let's now at the first search result:

```
>>> record[0]["LinkSetDb"][0]["Link"][0]
{'Id': '19304878'}
```

This is the article we searched for, which doesn't help us much, so let's look at the second search result:

```
>>> record[0]["LinkSetDb"][0]["Link"][1]
{'Id': '14630660'}
```

This paper, with PubMed ID 14630660, is about the Biopython PDB parser.

We can use a loop to print out all PubMed IDs:

```
>>> for link in record[0]["LinkSetDb"][0]["Link"]:
...     print(link["Id"])
...
19304878
14630660
18689808
17121776
16377612
12368254
......
```

Now that was nice, but personally I am often more interested to find out if a paper has been cited. Well, ELink can do that too – at least for journals in Pubmed Central (see Section 12.16.3).

For help on ELink, see the ELink help page. There is an entire sub-page just for the link names, describing how different databases can be cross referenced.

12.8 EGQuery: Global Query - counts for search terms

EGQuery provides counts for a search term in each of the Entrez databases (i.e. a global query). This is particularly useful to find out how many items your search terms would find in each database without actually performing lots of separate searches with ESearch (see the example in 12.15.2 below).

In this example, we use `Bio.Entrez.egquery()` to obtain the counts for "Biopython":

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="biopython")
>>> record = Entrez.read(stream)
```

```
>>> for row in record["eGQueryResult"]:
...     print(row["DbName"], row["Count"])
...
pubmed 6
pmc 62
journals 0
...
```

See the EGQuery help page for more information.

12.9 ESpell: Obtaining spelling suggestions

ESpell retrieves spelling suggestions. In this example, we use `Bio.Entrez.espell()` to obtain the correct spelling of Biopython:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.espell(term="biopythooon")
>>> record = Entrez.read(stream)
>>> record["Query"]
'biopythooon'
>>> record["CorrectedQuery"]
'biopython'
```

See the ESpell help page for more information. The main use of this is for GUI tools to provide automatic suggestions for search terms.

12.10 Parsing huge Entrez XML files

The `Entrez.read` function reads the entire XML file returned by Entrez into a single Python object, which is kept in memory. To parse Entrez XML files too large to fit in memory, you can use the function `Entrez.parse`. This is a generator function that reads records in the XML file one by one. This function is only useful if the XML file reflects a Python list object (in other words, if `Entrez.read` on a computer with infinite memory resources would return a Python list).

For example, you can download the entire Entrez Gene database for a given organism as a file from NCBI's ftp site. These files can be very large. As an example, on September 4, 2009, the file `Homo_sapiens.ags.gz`, containing the Entrez Gene database for human, had a size of 116576 kB. This file, which is in the `ASN` format, can be converted into an XML file using NCBI's `gene2xml` program (see NCBI's ftp site for more information):

```
$ gene2xml -b T -i Homo_sapiens.ags -o Homo_sapiens.xml
```

The resulting XML file has a size of 6.1 GB. Attempting `Entrez.read` on this file will result in a `MemoryError` on many computers.

The XML file `Homo_sapiens.xml` consists of a list of Entrez gene records, each corresponding to one Entrez gene in human. `Entrez.parse` retrieves these gene records one by one. You can then print out or store the relevant information in each record by iterating over the records. For example, this script iterates over the Entrez gene records and prints out the gene numbers and names for all current genes:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = open("Homo_sapiens.xml", "rb")
>>> records = Entrez.parse(stream)
```

Alternatively, you can use

```
>>> records = Entrez.parse("Homo_sapiens.xml")
```

and let `Bio.Entrez` take care of opening and closing the file. This is safer, as the file will then automatically be closed after parsing it, or if an error occurs.

```
>>> for record in records:
...     status = record["Entrezgene_track-info"]["Gene-track"]["Gene-track_status"]
...     if status.attributes["value"] == "discontinued":
...         continue
...     geneid = record["Entrezgene_track-info"]["Gene-track"]["Gene-track_geneid"]
...     genename = record["Entrezgene_gene"]["Gene-ref"]["Gene-ref_locus"]
...     print(geneid, genename)
...
1 A1BG
2 A2M
3 A2MP
8 AA
9 NAT1
10 NAT2
11 AACP
12 SERPINA3
13 AADAC
14 AAMP
15 AANAT
16 AARS
17 AAVS1
...
```

12.11 HTML escape characters

Pubmed records may contain HTML tags to indicate e.g. subscripts, superscripts, or italic text, as well as mathematical symbols via MathML. By default, the `Bio.Entrez` parser treats all text as plain text without markup; for example, the fragment "$P < 0.05$" in the abstract of a Pubmed record, which is encoded as

```
<i>P</i> &lt; 0.05
```

in the XML returned by Entrez, is converted to the Python string

```
'<i>P</i> < 0.05'
```

by the `Bio.Entrez` parser. While this is more human-readable, it is not valid HTML due to the less-than sign, and makes further processing of the text e.g. by an HTML parser impractical. To ensure that all strings returned by the parser are valid HTML, call `Entrez.read` or `Entrez.parse` with the `escape` argument set to `True`:

```
>>> record = Entrez.read(stream, escape=True)
```

The parser will then replace all characters disallowed in HTML by their HTML-escaped equivalent; in the example above, the parser will generate

```
'<i>P</i> &lt; 0.05'
```

which is a valid HTML fragment. By default, `escape` is `False`.

12.12 Handling errors

The file is not an XML file

For example, this error occurs if you try to parse a Fasta file as if it were an XML file:

```
>>> from Bio import Entrez
>>> stream = open("NC_005816.fna", "rb")  # a Fasta file
>>> record = Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.NotXMLError: Failed to parse the XML data (syntax error: line 1, column 0). Please mal
```

Here, the parser didn't find the `<?xml ...` tag with which an XML file is supposed to start, and therefore decides (correctly) that the file is not an XML file.

The file ends prematurely or is otherwise corrupted

When your file is in the XML format but is corrupted (for example, by ending prematurely), the parser will raise a CorruptedXMLError.

Here is an example of an XML file that ends prematurely:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.ncbi.nlm.nih.gov/e
<eInfoResult>
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nucleotide</DbName>
        <DbName>nuccore</DbName>
        <DbName>nucgss</DbName>
        <DbName>nucest</DbName>
        <DbName>structure</DbName>
        <DbName>genome</DbName>
        <DbName>books</DbName>
        <DbName>cancerchromosomes</DbName>
        <DbName>cdd</DbName>
```

which will generate the following traceback:

```
>>> Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found: line 16, column 0)
```

Note that the error message tells you at what point in the XML file the error was detected.

The file contains items that are missing from the associated DTD

This is an example of an XML file containing tags that do not have a description in the corresponding DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD eInfoResult, 11 May 2002//EN" "https://www.ncbi.nlm.nih.gov/e
```

```
<eInfoResult>
        <DbInfo>
        <DbName>pubmed</DbName>
        <MenuName>PubMed</MenuName>
        <Description>PubMed bibliographic record</Description>
        <Count>20161961</Count>
        <LastUpdate>2010/09/10 04:52</LastUpdate>
        <FieldList>
                <Field>
...
                </Field>
        </FieldList>
        <DocsumList>
                <Docsum>
                        <DsName>PubDate</DsName>
                        <DsType>4</DsType>
                        <DsTypeName>string</DsTypeName>
                </Docsum>
                <Docsum>
                        <DsName>EPubDate</DsName>
...
        </DbInfo>
</eInfoResult>
```

In this file, for some reason the tag `<DocsumList>` (and several others) are not listed in the DTD file `eInfo_020511.dtd`, which is specified on the second line as the DTD for this XML file. By default, the parser will stop and raise a ValidationError if it cannot find some tag in the DTD:

```
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
  ...
Bio.Entrez.Parser.ValidationError: Failed to find tag 'DocsumList' in the DTD. To skip all tags that ar
```

Optionally, you can instruct the parser to skip such tags instead of raising a ValidationError. This is done by calling `Entrez.read` or `Entrez.parse` with the argument `validate` equal to False:

```
>>> from Bio import Entrez
>>> stream = open("einfo3.xml", "rb")
>>> record = Entrez.read(stream, validate=False)
>>> stream.close()
```

Of course, the information contained in the XML tags that are not in the DTD are not present in the record returned by `Entrez.read`.

The file contains an error message

This may occur, for example, when you attempt to access a PubMed record for a nonexistent PubMed ID. By default, this will raise a `RuntimeError`:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
```

```
>>> stream = Entrez.esummary(db="pubmed", id="99999999")
>>> record = Entrez.read(stream)
Traceback (most recent call last):
...
RuntimeError: UID=99999999: cannot get document summary
```

If you are accessing multiple PubMed records, the `RuntimeError` would prevent you from receiving results for any of the PubMed records if one of the PubMed IDs is incorrect. To circumvent this, you can set the `ignore_errors` argument to `True`. This will return the requested results for the valid PubMed IDs, and an `ErrorElement` for the incorrect ID:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esummary(db="pubmed", id="19304878,99999999,31278684")
>>> record = Entrez.read(stream, ignore_errors=True)
>>> len(record)
3
>>> record[0].tag
'DocSum'
>>> record[0]["Title"]
'Biopython: freely available Python tools for computational molecular biology and bioinformatics.'
>>> record[1].tag
'ERROR'
>>> record[1]
ErrorElement('UID=99999999: cannot get document summary')
>>> record[2].tag
'DocSum'
>>> record[2]["Title"]
'Sharing Programming Resources Between Bio* Projects.'
```

12.13 Specialized parsers

The `Bio.Entrez.read()` function can parse most (if not all) XML output returned by Entrez. Entrez typically allows you to retrieve records in other formats, which may have some advantages compared to the XML format in terms of readability (or download size).

To request a specific file format from Entrez using `Bio.Entrez.efetch()` requires specifying the `rettype` and/or `retmode` optional arguments. The different combinations are described for each database type on the NCBI efetch webpage.

One obvious case is you may prefer to download sequences in the FASTA or GenBank/GenPept plain text formats (which can then be parsed with `Bio.SeqIO`, see Sections 5.3.1 and 12.6). For the literature databases, Biopython contains a parser for the `MEDLINE` format used in PubMed.

12.13.1 Parsing Medline records

You can find the Medline parser in `Bio.Medline`. Suppose we want to parse the file `pubmed_result1.txt`, containing one Medline record. You can find this file in Biopython's `Tests\Medline` directory. The file looks like this:

```
PMID- 12230038
OWN - NLM
STAT- MEDLINE
DA  - 20020916
```

```
DCOM- 20030606
LR  - 20041117
PUBM- Print
IS  - 1467-5463 (Print)
VI  - 3
IP  - 3
DP  - 2002 Sep
TI  - The Bio* toolkits--a brief overview.
PG  - 296-302
AB  - Bioinformatics research is often difficult to do with commercial software. The
      Open Source BioPerl, BioPython and Biojava projects provide toolkits with
...
```

We first open the file and then parse it:

```
>>> from Bio import Medline
>>> with open("pubmed_result1.txt") as stream:
...     record = Medline.read(stream)
...
```

The `record` now contains the Medline record as a Python dictionary:

```
>>> record["PMID"]
'12230038'

>>> record["AB"]
'Bioinformatics research is often difficult to do with commercial software.
The Open Source BioPerl, BioPython and Biojava projects provide toolkits with
multiple functionality that make it easier to create customized pipelines or
analysis. This review briefly compares the quirks of the underlying languages
and the functionality, documentation, utility and relative advantages of the
Bio counterparts, particularly from the point of view of the beginning
biologist programmer.'
```

The key names used in a Medline record can be rather obscure; use

```
>>> help(record)
```

for a brief summary.

To parse a file containing multiple Medline records, you can use the `parse` function instead:

```
>>> from Bio import Medline
>>> with open("pubmed_result2.txt") as stream:
...     for record in Medline.parse(stream):
...         print(record["TI"])
...
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
```

Instead of parsing Medline records stored in files, you can also parse Medline records downloaded by `Bio.Entrez.efetch`. For example, let's look at all Medline records in PubMed related to Biopython:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="biopython")
>>> record = Entrez.read(stream)
>>> record["IdList"]
['19304878', '18606172', '16403221', '16377612', '14871861', '14630660', '12230038']
```

We now use `Bio.Entrez.efetch` to download these Medline records:

```
>>> idlist = record["IdList"]
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
```

Here, we specify `rettype="medline", retmode="text"` to obtain the Medline records in plain-text Medline format. Now we use `Bio.Medline` to parse these records:

```
>>> from Bio import Medline
>>> records = Medline.parse(stream)
>>> for record in records:
...     print(record["AU"])
...
['Cock PJ', 'Antao T', 'Chang JT', 'Chapman BA', 'Cox CJ', 'Dalke A', ..., 'de Hoon MJ']
['Munteanu CR', 'Gonzalez-Diaz H', 'Magalhaes AL']
['Casbon JA', 'Crooks GE', 'Saqi MA']
['Pritchard L', 'White JA', 'Birch PR', 'Toth IK']
['de Hoon MJ', 'Imoto S', 'Nolan J', 'Miyano S']
['Hamelryck T', 'Manderick B']
['Mangalam H']
```

For comparison, here we show an example using the XML format:

```
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="xml")
>>> records = Entrez.read(stream)
>>> for record in records["PubmedArticle"]:
...     print(record["MedlineCitation"]["Article"]["ArticleTitle"])
...
Biopython: freely available Python tools for computational molecular biology and
 bioinformatics.
Enzymes/non-enzymes classification model complexity based on composition, sequence,
 3D and topological indices.
A high level interface to SCOP and ASTRAL implemented in python.
GenomeDiagram: a python package for the visualization of large-scale genomic data.
Open source clustering software.
PDB file parser and structure class implemented in Python.
The Bio* toolkits--a brief overview.
```

Note that in both of these examples, for simplicity we have naively combined ESearch and EFetch. In this situation, the NCBI would expect you to use their history feature, as illustrated in Section 12.16.

12.13.2 Parsing GEO records

GEO (Gene Expression Omnibus) is a data repository of high-throughput gene expression and hybridization array data. The `Bio.Geo` module can be used to parse GEO-formatted data.

The following code fragment shows how to parse the example GEO file `GSE16.txt` into a record and print the record:

```
>>> from Bio import Geo
>>> stream = open("GSE16.txt")
>>> records = Geo.parse(stream)
>>> for record in records:
...     print(record)
...
```

You can search the "gds" database (GEO datasets) with ESearch:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="gds", term="GSE16")
>>> record = Entrez.read(stream)
>>> stream.close()
>>> record["Count"]
'27'

>>> record["IdList"]
['200000016', '100000028', ...]
```

From the Entrez website, UID "200000016" is GDS16 while the other hit "100000028" is for the associated platform, GPL28. Unfortunately, at the time of writing the NCBI don't seem to support downloading GEO files using Entrez (not as XML, nor in the *Simple Omnibus Format in Text* (SOFT) format).

However, it is actually pretty straight forward to download the GEO files by FTP from ftp://ftp.ncbi.nih.gov/pub/geo/ instead. In this case you might want ftp://ftp.ncbi.nih.gov/pub/geo/DATA/SOFT/by_series/GSE16/GSE16_family.soft.gz (a compressed file, see the Python module gzip).

12.13.3 Parsing UniGene records

UniGene is an NCBI database of the transcriptome, with each UniGene record showing the set of transcripts that are associated with a particular gene in a specific organism. A typical UniGene record looks like this:

```
ID          Hs.2
TITLE       N-acetyltransferase 2 (arylamine N-acetyltransferase)
GENE        NAT2
CYTOBAND    8p22
GENE_ID     10
LOCUSLINK   10
HOMOL       YES
EXPRESS      bone| connective tissue| intestine| liver| liver tumor| normal| soft tissue/muscle tissue
RESTR_EXPR   adult
CHROMOSOME  8
STS         ACC=PMC310725P3 UNISTS=272646
STS         ACC=WIAF-2120 UNISTS=44576
STS         ACC=G59899 UNISTS=137181
...
STS         ACC=GDB:187676 UNISTS=155563
PROTSIM     ORG=10090; PROTGI=6754794; PROTID=NP_035004.1; PCT=76.55; ALN=288
PROTSIM     ORG=9796; PROTGI=149742490; PROTID=XP_001487907.1; PCT=79.66; ALN=288
PROTSIM     ORG=9986; PROTGI=126722851; PROTID=NP_001075655.1; PCT=76.90; ALN=288
...
PROTSIM     ORG=9598; PROTGI=114619004; PROTID=XP_519631.2; PCT=98.28; ALN=288
```

```
SCOUNT      38
SEQUENCE    ACC=BC067218.1; NID=g45501306; PID=g45501307; SEQTYPE=mRNA
SEQUENCE    ACC=NM_000015.2; NID=g116295259; PID=g116295260; SEQTYPE=mRNA
SEQUENCE    ACC=D90042.1; NID=g219415; PID=g219416; SEQTYPE=mRNA
SEQUENCE    ACC=D90040.1; NID=g219411; PID=g219412; SEQTYPE=mRNA
SEQUENCE    ACC=BC015878.1; NID=g16198419; PID=g16198420; SEQTYPE=mRNA
SEQUENCE    ACC=CR407631.1; NID=g47115198; PID=g47115199; SEQTYPE=mRNA
SEQUENCE    ACC=BG569293.1; NID=g13576946; CLONE=IMAGE:4722596; END=5'; LID=6989; SEQTYPE=EST; TRACE=44
...
SEQUENCE    ACC=AU099534.1; NID=g13550663; CLONE=HSI08034; END=5'; LID=8800; SEQTYPE=EST
//
```

This particular record shows the set of transcripts (shown in the `SEQUENCE` lines) that originate from the human gene NAT2, encoding en N-acetyltransferase. The `PROTSIM` lines show proteins with significant similarity to NAT2, whereas the `STS` lines show the corresponding sequence-tagged sites in the genome.

To parse UniGene files, use the `Bio.UniGene` module:

```
>>> from Bio import UniGene
>>> input = open("myunigenefile.data")
>>> record = UniGene.read(input)
```

The `record` returned by `UniGene.read` is a Python object with attributes corresponding to the fields in the UniGene record. For example,

```
>>> record.ID
"Hs.2"
>>> record.title
"N-acetyltransferase 2 (arylamine N-acetyltransferase)"
```

The `EXPRESS` and `RESTR_EXPR` lines are stored as Python lists of strings:

```
[
    "bone",
    "connective tissue",
    "intestine",
    "liver",
    "liver tumor",
    "normal",
    "soft tissue/muscle tissue tumor",
    "adult",
]
```

Specialized objects are returned for the `STS`, `PROTSIM`, and `SEQUENCE` lines, storing the keys shown in each line as attributes:

```
>>> record.sts[0].acc
'PMC310725P3'
>>> record.sts[0].unists
'272646'
```

and similarly for the `PROTSIM` and `SEQUENCE` lines.

To parse a file containing more than one UniGene record, use the `parse` function in `Bio.UniGene`:

```
>>> from Bio import UniGene
>>> input = open("unigenerecords.data")
```

```
>>> records = UniGene.parse(input)
>>> for record in records:
...     print(record.ID)
...
```

12.14 Using a proxy

Normally you won't have to worry about using a proxy, but if this is an issue on your network here is how to deal with it. Internally, `Bio.Entrez` uses the standard Python library `urllib` for accessing the NCBI servers. This will check an environment variable called `http_proxy` to configure any simple proxy automatically. Unfortunately this module does not support the use of proxies which require authentication.

You may choose to set the `http_proxy` environment variable once (how you do this will depend on your operating system). Alternatively you can set this within Python at the start of your script, for example:

```
import os

os.environ["http_proxy"] = "http://proxyhost.example.com:8080"
```

See the urllib documentation for more details.

12.15 Examples

12.15.1 PubMed and Medline

If you are in the medical field or interested in human issues (and many times even if you are not!), PubMed (https://www.ncbi.nlm.nih.gov/PubMed/) is an excellent source of all kinds of goodies. So like other things, we'd like to be able to grab information from it and use it in Python scripts.

In this example, we will query PubMed for all articles having to do with orchids (see section 2.3 for our motivation). We first check how many of such articles there are:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="orchid")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "pubmed":
...         print(row["Count"])
...
463
```

Now we use the `Bio.Entrez.efetch` function to download the PubMed IDs of these 463 articles:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(stream)
>>> stream.close()
>>> idlist = record["IdList"]
```

This returns a Python list containing all of the PubMed IDs of articles related to orchids:

```
>>> print(idlist)
['18680603', '18665331', '18661158', '18627489', '18627452', '18612381',
'18594007', '18591784', '18589523', '18579475', '18575811', '18575690',
...
```

Now that we've got them, we obviously want to get the corresponding Medline records and extract the information from them. Here, we'll download the Medline records in the Medline flat-file format, and use the `Bio.Medline` module to parse them:

```
>>> from Bio import Medline
>>> stream = Entrez.efetch(db="pubmed", id=idlist, rettype="medline", retmode="text")
>>> records = Medline.parse(stream)
```

NOTE - We've just done a separate search and fetch here, the NCBI much prefer you to take advantage of their history support in this situation. See Section 12.16.

Keep in mind that `records` is an iterator, so you can iterate through the records only once. If you want to save the records, you can convert them to a list:

```
>>> records = list(records)
```

Let's now iterate over the records to print out some information about each record:

```
>>> for record in records:
...     print("title:", record.get("TI", "?"))
...     print("authors:", record.get("AU", "?"))
...     print("source:", record.get("SO", "?"))
...     print("")
...
```

The output for this looks like:

```
title: Sex pheromone mimicry in the early spider orchid (ophrys sphegodes):
patterns of hydrocarbons as the key mechanism for pollination by sexual
deception [In Process Citation]
authors: ['Schiestl FP', 'Ayasse M', 'Paulus HF', 'Lofstedt C', 'Hansson BS',
'Ibarra F', 'Francke W']
source: J Comp Physiol [A] 2000 Jun;186(6):567-74
```

Especially interesting to note is the list of authors, which is returned as a standard Python list. This makes it easy to manipulate and search using standard Python tools. For instance, we could loop through a whole bunch of entries searching for a particular author with code like the following:

```
>>> search_author = "Waits T"
>>> for record in records:
...     if not "AU" in record:
...         continue
...     if search_author in record["AU"]:
...         print("Author %s found: %s" % (search_author, record["SO"]))
...
```

Hopefully this section gave you an idea of the power and flexibility of the Entrez and Medline interfaces and how they can be used together.

12.15.2 Searching, downloading, and parsing Entrez Nucleotide records

Here we'll show a simple example of performing a remote Entrez query. In section 2.3 of the parsing examples, we talked about using NCBI's Entrez website to search the NCBI nucleotide databases for info on Cypripedioideae, our friends the lady slipper orchids. Now, we'll look at how to automate that process using a Python script. In this example, we'll just show how to connect, get the results, and parse them, with the Entrez module doing all of the work.

First, we use EGQuery to find out the number of results we will get before actually downloading them. EGQuery will tell us how many search results were found in each of the databases, but for this example we are only interested in nucleotides:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "nuccore":
...         print(row["Count"])
...
4457
```

So, we expect to find 4457 Entrez Nucleotide records (this increased from 814 records in 2008; it is likely to continue to increase in the future). If you find some ridiculously high number of hits, you may want to reconsider if you really want to download all of them, which is our next step. Let's use the `retmax` argument to restrict the maximum number of records retrieved to the number available in 2008:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Cypripedioideae", retmax=814, idtype="acc"
... )
>>> record = Entrez.read(stream)
>>> stream.close()
```

Here, `record` is a Python dictionary containing the search results and some auxiliary information. Just for information, let's look at what is stored in this dictionary:

```
>>> print(record.keys())
['Count', 'RetMax', 'IdList', 'TranslationSet', 'RetStart', 'QueryTranslation']
```

First, let's check how many results were found:

```
>>> print(record["Count"])
'4457'
```

You might have expected this to be 814, the maximum number of records we asked to retrieve. However, `Count` represents the total number of records available for that search, not how many were retrieved. The retrieved records are stored in `record['IdList']`, which should contain the total number we asked for:

```
>>> len(record["IdList"])
814
```

Let's look at the first five results:

```
>>> record["IdList"][:5]
['KX265015.1', 'KX265014.1', 'KX265013.1', 'KX265012.1', 'KX265011.1']
```

We can download these records using `efetch`. While you could download these records one by one, to reduce the load on NCBI's servers, it is better to fetch a bunch of records at the same time, shown below. However, in this situation you should ideally be using the history feature described later in Section 12.16.

```
>>> idlist = ",".join(record["IdList"][:5])
>>> print(idlist)
KX265015.1, KX265014.1, KX265013.1, KX265012.1, KX265011.1]
>>> stream = Entrez.efetch(db="nucleotide", id=idlist, retmode="xml")
>>> records = Entrez.read(stream)
>>> len(records)
5
```

Each of these records corresponds to one GenBank record.

```
>>> print(records[0].keys())
['GBSeq_moltype', 'GBSeq_source', 'GBSeq_sequence',
 'GBSeq_primary-accession', 'GBSeq_definition', 'GBSeq_accession-version',
 'GBSeq_topology', 'GBSeq_length', 'GBSeq_feature-table',
 'GBSeq_create-date', 'GBSeq_other-seqids', 'GBSeq_division',
 'GBSeq_taxonomy', 'GBSeq_references', 'GBSeq_update-date',
 'GBSeq_organism', 'GBSeq_locus', 'GBSeq_strandedness']

>>> print(records[0]["GBSeq_primary-accession"])
DQ110336

>>> print(records[0]["GBSeq_other-seqids"])
['gb|DQ110336.1|', 'gi|187237168']

>>> print(records[0]["GBSeq_definition"])
Cypripedium calceolus voucher Davis 03-03 A maturase (matR) gene, partial cds;
mitochondrial

>>> print(records[0]["GBSeq_organism"])
Cypripedium calceolus
```

You could use this to quickly set up searches – but for heavy usage, see Section 12.16.

12.15.3 Searching, downloading, and parsing GenBank records

The GenBank record format is a very popular method of holding information about sequences, sequence features, and other associated sequence information. The format is a good way to get information from the NCBI databases at https://www.ncbi.nlm.nih.gov/.

In this example we'll show how to query the NCBI databases,to retrieve the records from the query, and then parse them using `Bio.SeqIO` - something touched on in Section 5.3.1. For simplicity, this example *does not* take advantage of the WebEnv history feature – see Section 12.16 for this.

First, we want to make a query and find out the ids of the records to retrieve. Here we'll do a quick search for one of our favorite organisms, *Opuntia* (prickly-pear cacti). We can do quick search and get back the GIs (GenBank identifiers) for all of the corresponding records. First we check how many records there are:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.egquery(term="Opuntia AND rpl16")
```

```
>>> record = Entrez.read(stream)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"] == "nuccore":
...         print(row["Count"])
...
9
```

Now we download the list of GenBank identifiers:

```
>>> stream = Entrez.esearch(db="nuccore", term="Opuntia AND rpl16")
>>> record = Entrez.read(stream)
>>> gi_list = record["IdList"]
>>> gi_list
['57240072', '57240071', '6273287', '6273291', '6273290', '6273289', '6273286',
'6273285', '6273284']
```

Now we use these GIs to download the GenBank records - note that with older versions of Biopython you had to supply a comma separated list of GI numbers to Entrez, as of Biopython 1.59 you can pass a list and this is converted for you:

```
>>> gi_str = ",".join(gi_list)
>>> stream = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
```

If you want to look at the raw GenBank files, you can read from this stream and print out the result:

```
>>> text = stream.read()
>>> print(text)
LOCUS       AY851612                 892 bp    DNA     linear   PLN 10-APR-2007
DEFINITION  Opuntia subulata rpl16 gene, intron; chloroplast.
ACCESSION   AY851612
VERSION     AY851612.1  GI:57240072
KEYWORDS    .
SOURCE      chloroplast Austrocylindropuntia subulata
  ORGANISM  Austrocylindropuntia subulata
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
            Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;
            Caryophyllales; Cactaceae; Opuntioideae; Austrocylindropuntia.
REFERENCE   1  (bases 1 to 892)
  AUTHORS   Butterworth,C.A. and Wallace,R.S.
...
```

In this case, we are just getting the raw records. To get the records in a more Python-friendly form, we can use `Bio.SeqIO` to parse the GenBank data into `SeqRecord` objects, including `SeqFeature` objects (see Chapter 5):

```
>>> from Bio import SeqIO
>>> stream = Entrez.efetch(db="nuccore", id=gi_str, rettype="gb", retmode="text")
>>> records = SeqIO.parse(stream, "gb")
```

We can now step through the records and look at the information we are interested in:

```
>>> for record in records:
...     print(f"{record.name}, length {len(record)}, with {len(record.features)} features")
...
```

```
AY851612, length 892, with 3 features
AY851611, length 881, with 3 features
AF191661, length 895, with 3 features
AF191665, length 902, with 3 features
AF191664, length 899, with 3 features
AF191663, length 899, with 3 features
AF191660, length 893, with 3 features
AF191659, length 894, with 3 features
AF191658, length 896, with 3 features
```

Using these automated query retrieval functionality is a big plus over doing things by hand. Although the module should obey the NCBI's max three queries per second rule, the NCBI have other recommendations like avoiding peak hours. See Section 12.1. In particular, please note that for simplicity, this example does not use the WebEnv history feature. You should use this for any non-trivial search and download work, see Section 12.16.

Finally, if plan to repeat your analysis, rather than downloading the files from the NCBI and parsing them immediately (as shown in this example), you should just download the records *once* and save them to your hard disk, and then parse the local file.

12.15.4 Finding the lineage of an organism

Staying with a plant example, let's now find the lineage of the Cypripedioideae orchid family. First, we search the Taxonomy database for Cypripedioideae, which yields exactly one NCBI taxonomy identifier:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(db="Taxonomy", term="Cypripedioideae")
>>> record = Entrez.read(stream)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

Now, we use `efetch` to download this entry in the Taxonomy database, and then parse it:

```
>>> stream = Entrez.efetch(db="Taxonomy", id="158330", retmode="xml")
>>> records = Entrez.read(stream)
```

Again, this record stores lots of information:

```
>>> records[0].keys()
['Lineage', 'Division', 'ParentTaxId', 'PubDate', 'LineageEx',
 'CreateDate', 'TaxId', 'Rank', 'GeneticCode', 'ScientificName',
 'MitoGeneticCode', 'UpdateDate']
```

We can get the lineage directly from this record:

```
>>> records[0]["Lineage"]
'cellular organisms; Eukaryota; Viridiplantae; Streptophyta; Streptophytina;
 Embryophyta; Tracheophyta; Euphyllophyta; Spermatophyta; Magnoliopsida;
 Liliopsida; Asparagales; Orchidaceae'
```

The record data contains much more than just the information shown here - for example look under `"LineageEx"` instead of `"Lineage"` and you'll get the NCBI taxon identifiers of the lineage entries too.

12.16 Using the history and WebEnv

Often you will want to make a series of linked queries. Most typically, running a search, perhaps refining the search, and then retrieving detailed search results. You *can* do this by making a series of separate calls to Entrez. However, the NCBI prefer you to take advantage of their history support - for example combining ESearch and EFetch.

Another typical use of the history support would be to combine EPost and EFetch. You use EPost to upload a list of identifiers, which starts a new history session. You then download the records with EFetch by referring to the session (instead of the identifiers).

12.16.1 Searching for and downloading sequences using the history

Suppose we want to search and download all the *Opuntia* rpl16 nucleotide sequences, and store them in a FASTA file. As shown in Section 12.15.3, we can naively combine `Bio.Entrez.esearch()` to get a list of Accession numbers, and then call `Bio.Entrez.efetch()` to download them all.

However, the approved approach is to run the search with the history feature. Then, we can fetch the results by reference to the search results - which the NCBI can anticipate and cache.

To do this, call `Bio.Entrez.esearch()` as normal, but with the additional argument of `usehistory="y"`,

```
>>> from Bio import Entrez
>>> Entrez.email = "history.user@example.com"  # Always tell NCBI who you are
>>> stream = Entrez.esearch(
...     db="nucleotide", term="Opuntia[orgn] and rpl16", usehistory="y", idtype="acc"
... )
>>> search_results = Entrez.read(stream)
>>> stream.close()
```

As before (see Section 12.15.2), the XML output includes the first `retmax` search results, with `retmax` defaulting to 20:

```
>>> acc_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> len(acc_list)
20

>>> count
28
```

You also get given two additional pieces of information, the `WebEnv` session cookie, and the `QueryKey`:

```
>>> webenv = search_results["WebEnv"]
>>> query_key = search_results["QueryKey"]
```

Having stored these values in variables `session_cookie` and `query_key` we can use them as parameters to `Bio.Entrez.efetch()` instead of giving the GI numbers as identifiers.

While for small searches you might be OK downloading everything at once, it is better to download in batches. You use the `retstart` and `retmax` parameters to specify which range of search results you want returned (starting entry using zero-based counting, and maximum number of results to return). Note that if Biopython encounters a transient failure like a HTTP 500 response when communicating with NCBI, it will automatically try again a couple of times. For example,

```
# This assumes you have already run a search as shown above,
# and set the variables count, webenv, query_key
```

```
batch_size = 3
output = open("orchid_rpl16.fasta", "w")
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="nucleotide",
        rettype="fasta",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=webenv,
        query_key=query_key,
        idtype="acc",
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()
```

For illustrative purposes, this example downloaded the FASTA records in batches of three. Unless you are downloading genomes or chromosomes, you would normally pick a larger batch size.

12.16.2 Searching for and downloading abstracts using the history

Here is another history example, searching for papers published in the last year about the *Opuntia*, and then downloading them into a file in MedLine format:

```
from Bio import Entrez

Entrez.email = "history.user@example.com"
search_results = Entrez.read(
    Entrez.esearch(
        db="pubmed", term="Opuntia[ORGN]", reldate=365, datetype="pdat", usehistory="y"
    )
)
count = int(search_results["Count"])
print("Found %i results" % count)

batch_size = 10
output = open("recent_orchid_papers.txt", "w")
for start in range(0, count, batch_size):
    end = min(count, start + batch_size)
    print("Going to download record %i to %i" % (start + 1, end))
    stream = Entrez.efetch(
        db="pubmed",
        rettype="medline",
        retmode="text",
        retstart=start,
        retmax=batch_size,
        webenv=search_results["WebEnv"],
        query_key=search_results["QueryKey"],
```

```
    )
    data = stream.read()
    stream.close()
    output.write(data)
output.close()
```

At the time of writing, this gave 28 matches - but because this is a date dependent search, this will of course vary. As described in Section 12.13.1 above, you can then use `Bio.Medline` to parse the saved records.

12.16.3 Searching for citations

Back in Section 12.7 we mentioned ELink can be used to search for citations of a given paper. Unfortunately this only covers journals indexed for PubMed Central (doing it for all the journals in PubMed would mean a lot more work for the NIH). Let's try this for the Biopython PDB parser paper, PubMed ID 14630660:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"  # Always tell NCBI who you are
>>> pmid = "14630660"
>>> results = Entrez.read(
...     Entrez.elink(dbfrom="pubmed", db="pmc", LinkName="pubmed_pmc_refs", id=pmid)
... )
>>> pmc_ids = [link["Id"] for link in results[0]["LinkSetDb"][0]["Link"]]
>>> pmc_ids
['2744707', '2705363', '2682512', ..., '1190160']
```

Great - eleven articles. But why hasn't the Biopython application note been found (PubMed ID 19304878)? Well, as you might have guessed from the variable names, there are not actually PubMed IDs, but PubMed Central IDs. Our application note is the third citing paper in that list, PMCID 2682512.

So, what if (like me) you'd rather get back a list of PubMed IDs? Well we can call ELink again to translate them. This becomes a two step process, so by now you should expect to use the history feature to accomplish it (Section 12.16).

But first, taking the more straightforward approach of making a second (separate) call to ELink:

```
>>> results2 = Entrez.read(
...     Entrez.elink(dbfrom="pmc", db="pubmed", LinkName="pmc_pubmed", id=",".join(pmc_ids))
... )
>>> pubmed_ids = [link["Id"] for link in results2[0]["LinkSetDb"][0]["Link"]]
>>> pubmed_ids
['19698094', '19450287', '19304878', ..., '15985178']
```

This time you can immediately spot the Biopython application note as the third hit (PubMed ID 19304878).

Now, let's do that all again but with the history ... *TODO*.

And finally, don't forget to include your *own* email address in the Entrez calls.

Chapter 13

Swiss-Prot and ExPASy

13.1 Parsing Swiss-Prot files

Swiss-Prot (https://web.expasy.org/docs/swiss-prot_guideline.html) is a hand-curated database of protein sequences. Biopython can parse the "plain text" Swiss-Prot file format, which is still used for the UniProt Knowledgebase which combined Swiss-Prot, TrEMBL and PIR-PSD.

Although in the following we focus on the older human readable plain text format, `Bio.SeqIO` can read both this and the newer UniProt XML file format for annotated protein sequences.

13.1.1 Parsing Swiss-Prot records

In Section 5.3.2, we described how to extract the sequence of a Swiss-Prot record as a `SeqRecord` object. Alternatively, you can store the Swiss-Prot record in a `Bio.SwissProt.Record` object, which in fact stores the complete information contained in the Swiss-Prot record. In this section, we describe how to extract `Bio.SwissProt.Record` objects from a Swiss-Prot file.

To parse a Swiss-Prot record, we first get a handle to a Swiss-Prot record. There are several ways to do so, depending on where and how the Swiss-Prot record is stored:

- Open a Swiss-Prot file locally:

  ```
  >>> handle = open("SwissProt/F2CXE6.txt")
  ```

- Open a gzipped Swiss-Prot file:

  ```
  >>> import gzip
  >>> handle = gzip.open("myswissprotfile.dat.gz", "rt")
  ```

- Open a Swiss-Prot file over the internet:

  ```
  >>> from urllib.request import urlopen
  >>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/SwissProt/F2CXE6.txt"
  >>> handle = urlopen(url)
  ```

  to open the file stored on the Internet before calling `read`.

- Open a Swiss-Prot file over the internet from the ExPASy database (see section 13.5.1):

  ```
  >>> from Bio import ExPASy
  >>> handle = ExPASy.get_sprot_raw("F2CXE6")
  ```

The key point is that for the parser, it doesn't matter how the handle was created, as long as it points to data in the Swiss-Prot format. The parser will automatically decode the data as ASCII (the encoding used by Swiss-Prot) if the handle was opened in binary mode.

We can use `Bio.SeqIO` as described in Section 5.3.2 to get file format agnostic `SeqRecord` objects. Alternatively, we can use `Bio.SwissProt` get `Bio.SwissProt.Record` objects, which are a much closer match to the underlying file format.

To read one Swiss-Prot record from the handle, we use the function `read()`:

```
>>> from Bio import SwissProt
>>> record = SwissProt.read(handle)
```

This function should be used if the handle points to exactly one Swiss-Prot record. It raises a `ValueError` if no Swiss-Prot record was found, and also if more than one record was found.

We can now print out some information about this record:

```
>>> print(record.description)
SubName: Full=Plasma membrane intrinsic protein {ECO:0000313|EMBL:BAN04711.1}; SubName: Full=Predicted p
>>> for ref in record.references:
...     print("authors:", ref.authors)
...     print("title:", ref.title)
...
authors: Matsumoto T., Tanaka T., Sakai H., Amano N., Kanamori H., Kurita K., Kikuta A., Kamiya K., Yama
title: Comprehensive sequence analysis of 24,783 barley full-length cDNAs derived from 12 clone librari
authors: Shibasaka M., Sasano S., Utsugi S., Katsuhara M.
title: Functional characterization of a novel plasma membrane intrinsic protein2 in barley.
authors: Shibasaka M., Katsuhara M., Sasano S.
title:
>>> print(record.organism_classification)
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', 'Tracheophyta', 'Spermatophyta', 'Magnoli
```

To parse a file that contains more than one Swiss-Prot record, we use the `parse` function instead. This function allows us to iterate over the records in the file.

For example, let's parse the full Swiss-Prot database and collect all the descriptions. You can download this from the ExPASy FTP site as a single gzipped-file `uniprot_sprot.dat.gz` (about 300MB). This is a compressed file containing a single file, `uniprot_sprot.dat` (over 1.5GB).

As described at the start of this section, you can use the Python library `gzip` to open and uncompress a `.gz` file, like this:

```
>>> import gzip
>>> handle = gzip.open("uniprot_sprot.dat.gz", "rt")
```

However, uncompressing a large file takes time, and each time you open the file for reading in this way, it has to be decompressed on the fly. So, if you can spare the disk space you'll save time in the long run if you first decompress the file to disk, to get the `uniprot_sprot.dat` file inside. Then you can open the file for reading as usual:

```
>>> handle = open("uniprot_sprot.dat")
```

As of June 2009, the full Swiss-Prot database downloaded from ExPASy contained 468851 Swiss-Prot records. One concise way to build up a list of the record descriptions is with a list comprehension:

```
>>> from Bio import SwissProt
>>> handle = open("uniprot_sprot.dat")
>>> descriptions = [record.description for record in SwissProt.parse(handle)]
```

```
>>> len(descriptions)
468851
>>> descriptions[:5]
['RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-1R;',
 'RecName: Full=Protein MGF 100-2L;']
```

Or, using a for loop over the record iterator:

```
>>> from Bio import SwissProt
>>> descriptions = []
>>> handle = open("uniprot_sprot.dat")
>>> for record in SwissProt.parse(handle):
...     descriptions.append(record.description)
...
>>> len(descriptions)
468851
```

Because this is such a large input file, either way takes about eleven minutes on my new desktop computer (using the uncompressed `uniprot_sprot.dat` file as input).

It is equally easy to extract any kind of information you'd like from Swiss-Prot records. To see the members of a Swiss-Prot record, use

```
>>> dir(record)
['__doc__', '__init__', '__module__', 'accessions', 'annotation_update',
'comments', 'created', 'cross_references', 'data_class', 'description',
'entry_name', 'features', 'gene_name', 'host_organism', 'keywords',
'molecule_type', 'organelle', 'organism', 'organism_classification',
'references', 'seqinfo', 'sequence', 'sequence_length',
'sequence_update', 'taxonomy_id']
```

13.1.2 Parsing the Swiss-Prot keyword and category list

Swiss-Prot also distributes a file `keywlist.txt`, which lists the keywords and categories used in Swiss-Prot. The file contains entries in the following form:

```
ID   2Fe-2S.
AC   KW-0001
DE   Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron
DE   atoms complexed to 2 inorganic sulfides and 4 sulfur atoms of
DE   cysteines from the protein.
SY   Fe2S2; [2Fe-2S] cluster; [Fe2S2] cluster; Fe2/S2 (inorganic) cluster;
SY   Di-mu-sulfido-diiron; 2 iron, 2 sulfur cluster binding.
GO   GO:0051537; 2 iron, 2 sulfur cluster binding
HI   Ligand: Iron; Iron-sulfur; 2Fe-2S.
HI   Ligand: Metal-binding; 2Fe-2S.
CA   Ligand.
//
ID   3D-structure.
AC   KW-0002
```

```
DE   Protein, or part of a protein, whose three-dimensional structure has
DE   been resolved experimentally (for example by X-ray crystallography or
DE   NMR spectroscopy) and whose coordinates are available in the PDB
DE   database. Can also be used for theoretical models.
HI   Technical term: 3D-structure.
CA   Technical term.
//
ID   3Fe-4S.
...
```

The entries in this file can be parsed by the `parse` function in the `Bio.SwissProt.KeyWList` module. Each entry is then stored as a `Bio.SwissProt.KeyWList.Record`, which is a Python dictionary.

```
>>> from Bio.SwissProt import KeyWList
>>> handle = open("keywlist.txt")
>>> records = KeyWList.parse(handle)
>>> for record in records:
...     print(record["ID"])
...     print(record["DE"])
...
```

This prints

```
2Fe-2S.
Protein which contains at least one 2Fe-2S iron-sulfur cluster: 2 iron atoms
complexed to 2 inorganic sulfides and 4 sulfur atoms of cysteines from the
protein.
...
```

13.2 Parsing Prosite records

Prosite is a database containing protein domains, protein families, functional sites, as well as the patterns and profiles to recognize them. Prosite was developed in parallel with Swiss-Prot. In Biopython, a Prosite record is represented by the `Bio.ExPASy.Prosite.Record` class, whose members correspond to the different fields in a Prosite record.

In general, a Prosite file can contain more than one Prosite records. For example, the full set of Prosite records, which can be downloaded as a single file (`prosite.dat`) from the ExPASy FTP site, contains 2073 records (version 20.24 released on 4 December 2007). To parse such a file, we again make use of an iterator:

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("myprositefile.dat")
>>> records = Prosite.parse(handle)
```

We can now take the records one at a time and print out some information. For example, using the file containing the complete Prosite database, we'd find

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> record = next(records)
>>> record.accession
'PS00001'
>>> record.name
```

```
'ASN_GLYCOSYLATION'
>>> record.pdoc
'PDOC00001'
>>> record = next(records)
>>> record.accession
'PS00004'
>>> record.name
'CAMP_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00004'
>>> record = next(records)
>>> record.accession
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

and so on. If you're interested in how many Prosite records there are, you could use

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> n = 0
>>> for record in records:
...     n += 1
...
>>> n
2073
```

To read exactly one Prosite from the handle, you can use the `read` function:

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("mysingleprositerecord.dat")
>>> record = Prosite.read(handle)
```

This function raises a ValueError if no Prosite record is found, and also if more than one Prosite record is found.

13.3 Parsing Prosite documentation records

In the Prosite example above, the `record.pdoc` accession numbers `'PDOC00001'`, `'PDOC00004'`, `'PDOC00005'` and so on refer to Prosite documentation. The Prosite documentation records are available from ExPASy as individual files, and as one file (`prosite.doc`) containing all Prosite documentation records.

We use the parser in `Bio.ExPASy.Prodoc` to parse Prosite documentation records. For example, to create a list of all accession numbers of Prosite documentation record, you can use

```
>>> from Bio.ExPASy import Prodoc
>>> handle = open("prosite.doc")
>>> records = Prodoc.parse(handle)
>>> accessions = [record.accession for record in records]
```

Again a `read()` function is provided to read exactly one Prosite documentation record from the handle.

13.4 Parsing Enzyme records

ExPASy's Enzyme database is a repository of information on enzyme nomenclature. A typical Enzyme record looks as follows:

```
ID   3.1.1.34
DE   Lipoprotein lipase.
AN   Clearing factor lipase.
AN   Diacylglycerol lipase.
AN   Diglyceride lipase.
CA   Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.
CC   -!- Hydrolyzes triacylglycerols in chylomicrons and very low-density
CC       lipoproteins (VLDL).
CC   -!- Also hydrolyzes diacylglycerol.
PR   PROSITE; PDOC00110;
DR   P11151, LIPL_BOVIN ;  P11153, LIPL_CAVPO ;  P11602, LIPL_CHICK ;
DR   P55031, LIPL_FELCA ;  P06858, LIPL_HUMAN ;  P11152, LIPL_MOUSE ;
DR   O46647, LIPL_MUSVI ;  P49060, LIPL_PAPAN ;  P49923, LIPL_PIG   ;
DR   Q06000, LIPL_RAT   ;  Q29524, LIPL_SHEEP ;
//
```

In this example, the first line shows the EC (Enzyme Commission) number of lipoprotein lipase (second line). Alternative names of lipoprotein lipase are "clearing factor lipase", "diacylglycerol lipase", and "diglyceride lipase" (lines 3 through 5). The line starting with "CA" shows the catalytic activity of this enzyme. Comment lines start with "CC". The "PR" line shows references to the Prosite Documentation records, and the "DR" lines show references to Swiss-Prot records. Not of these entries are necessarily present in an Enzyme record.

In Biopython, an Enzyme record is represented by the `Bio.ExPASy.Enzyme.Record` class. This record derives from a Python dictionary and has keys corresponding to the two-letter codes used in Enzyme files. To read an Enzyme file containing one Enzyme record, use the `read` function in `Bio.ExPASy.Enzyme`:

```
>>> from Bio.ExPASy import Enzyme
>>> with open("lipoprotein.txt") as handle:
...     record = Enzyme.read(handle)
...
>>> record["ID"]
'3.1.1.34'
>>> record["DE"]
'Lipoprotein lipase.'
>>> record["AN"]
['Clearing factor lipase.', 'Diacylglycerol lipase.', 'Diglyceride lipase.']
>>> record["CA"]
'Triacylglycerol + H(2)O = diacylglycerol + a carboxylate.'
>>> record["PR"]
['PDOC00110']
>>> record["CC"]
['Hydrolyzes triacylglycerols in chylomicrons and very low-density lipoproteins
(VLDL).', 'Also hydrolyzes diacylglycerol.']
>>> record["DR"]
[['P11151', 'LIPL_BOVIN'], ['P11153', 'LIPL_CAVPO'], ['P11602', 'LIPL_CHICK'],
['P55031', 'LIPL_FELCA'], ['P06858', 'LIPL_HUMAN'], ['P11152', 'LIPL_MOUSE'],
['O46647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'],
['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```

The `read` function raises a ValueError if no Enzyme record is found, and also if more than one Enzyme record is found.

The full set of Enzyme records can be downloaded as a single file (`enzyme.dat`) from the ExPASy FTP site, containing 4877 records (release of 3 March 2009). To parse such a file containing multiple Enzyme records, use the `parse` function in `Bio.ExPASy.Enzyme` to obtain an iterator:

```
>>> from Bio.ExPASy import Enzyme
>>> handle = open("enzyme.dat")
>>> records = Enzyme.parse(handle)
```

We can now iterate over the records one at a time. For example, we can make a list of all EC numbers for which an Enzyme record is available:

```
>>> ecnumbers = [record["ID"] for record in records]
```

13.5 Accessing the ExPASy server

Swiss-Prot, Prosite, and Prosite documentation records can be downloaded from the ExPASy web server at https://www.expasy.org. Four kinds of queries are available from ExPASy:

get_prodoc_entry To download a Prosite documentation record in HTML format

get_prosite_entry To download a Prosite record in HTML format

get_prosite_raw To download a Prosite or Prosite documentation record in raw format

get_sprot_raw To download a Swiss-Prot record in raw format

To access this web server from a Python script, we use the `Bio.ExPASy` module.

13.5.1 Retrieving a Swiss-Prot record

Let's say we are looking at chalcone synthases for Orchids (see section 2.3 for some justification for looking for interesting things about orchids). Chalcone synthase is involved in flavanoid biosynthesis in plants, and flavanoids make lots of cool things like pigment colors and UV protectants.

If you do a search on Swiss-Prot, you can find three orchid proteins for Chalcone Synthase, id numbers O23729, O23730, O23731. Now, let's write a script which grabs these, and parses out some interesting information.

First, we grab the records, using the `get_sprot_raw()` function of `Bio.ExPASy`. This function is very nice since you can feed it an id and get back a handle to a raw text record (no HTML to mess with!). We can the use `Bio.SwissProt.read` to pull out the Swiss-Prot record, or `Bio.SeqIO.read` to get a SeqRecord. The following code accomplishes what I just wrote:

```
>>> from Bio import ExPASy
>>> from Bio import SwissProt

>>> accessions = ["O23729", "O23730", "O23731"]
>>> records = []

>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     record = SwissProt.read(handle)
...     records.append(record)
...
```

If the accession number you provided to `ExPASy.get_sprot_raw` does not exist, then `SwissProt.read(handle)` will raise a `ValueError`. You can catch `ValueException` exceptions to detect invalid accession numbers:

```
>>> for accession in accessions:
...     handle = ExPASy.get_sprot_raw(accession)
...     try:
...         record = SwissProt.read(handle)
...     except ValueException:
...         print("WARNING: Accession %s not found" % accession)
...     records.append(record)
...
```

13.5.2 Searching Swiss-Prot

Now, you may remark that I knew the records' accession numbers beforehand. Indeed, `get_sprot_raw()` needs either the entry name or an accession number. When you don't have them handy, right now you could use https://www.uniprot.org/ but we do not have a Python wrapper for searching this from a script. Perhaps you could contribute here?

13.5.3 Retrieving Prosite and Prosite documentation records

Prosite and Prosite documentation records can be retrieved either in HTML format, or in raw format. To parse Prosite and Prosite documentation records with Biopython, you should retrieve the records in raw format. For other purposes, however, you may be interested in these records in HTML format.

To retrieve a Prosite or Prosite documentation record in raw format, use `get_prosite_raw()`. For example, to download a Prosite record and print it out in raw text format, use

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_raw("PS00001")
>>> text = handle.read()
>>> print(text)
```

To retrieve a Prosite record and parse it into a `Bio.Prosite.Record` object, use

```
>>> from Bio import ExPASy
>>> from Bio import Prosite
>>> handle = ExPASy.get_prosite_raw("PS00001")
>>> record = Prosite.read(handle)
```

The same function can be used to retrieve a Prosite documentation record and parse it into a `Bio.ExPASy.Prodoc.Record` object:

```
>>> from Bio import ExPASy
>>> from Bio.ExPASy import Prodoc
>>> handle = ExPASy.get_prosite_raw("PDOC00001")
>>> record = Prodoc.read(handle)
```

For non-existing accession numbers, `ExPASy.get_prosite_raw` returns a handle to an empty string. When faced with an empty string, `Prosite.read` and `Prodoc.read` will raise a ValueError. You can catch these exceptions to detect invalid accession numbers.

The functions `get_prosite_entry()` and `get_prodoc_entry()` are used to download Prosite and Prosite documentation records in HTML format. To create a web page showing one Prosite record, you can use

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry("PS00001")
>>> html = handle.read()
>>> with open("myprositerecord.html", "w") as out_handle:
...     out_handle.write(html)
...
```

and similarly for a Prosite documentation record:

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry("PDOC00001")
>>> html = handle.read()
>>> with open("myprodocrecord.html", "w") as out_handle:
...     out_handle.write(html)
...
```

For these functions, an invalid accession number returns an error message in HTML format.

13.6 Scanning the Prosite database

ScanProsite allows you to scan protein sequences online against the Prosite database by providing a UniProt or PDB sequence identifier or the sequence itself. For more information about ScanProsite, please see the ScanProsite documentation as well as the documentation for programmatic access of ScanProsite.

You can use Biopython's `Bio.ExPASy.ScanProsite` module to scan the Prosite database from Python. This module both helps you to access ScanProsite programmatically, and to parse the results returned by ScanProsite. To scan for Prosite patterns in the following protein sequence:

```
MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT
CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN
```

you can use the following code:

```
>>> sequence = (
...     "MEHKEVVLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT"
...     "CRAFQYHSKEQQCVIMAENRKSSIIIRMRDVVLFEKKVYLSECKTGNGKNYRGTMSKTKN"
... )
>>> from Bio.ExPASy import ScanProsite
>>> handle = ScanProsite.scan(seq=sequence)
```

By executing `handle.read()`, you can obtain the search results in raw XML format. Instead, let's use `Bio.ExPASy.ScanProsite.read` to parse the raw XML into a Python object:

```
>>> result = ScanProsite.read(handle)
>>> type(result)
<class 'Bio.ExPASy.ScanProsite.Record'>
```

A `Bio.ExPASy.ScanProsite.Record` object is derived from a list, with each element in the list storing one ScanProsite hit. This object also stores the number of hits, as well as the number of search sequences, as returned by ScanProsite. This ScanProsite search resulted in six hits:

```
>>> result.n_seq
1
>>> result.n_match
```

```
1
>>> len(result)
1
>>> result[0]
{'sequence_ac': 'USERSEQ1', 'start': 16, 'stop': 98, 'signature_ac': 'PS50948', 'score': '8.873', 'level
```

Other ScanProsite parameters can be passed as keyword arguments; see the documentation for programmatic access of ScanProsite for more information. As an example, passing `lowscore=1` to include matches with low level scores lets use find one additional hit:

```
>>> handle = ScanProsite.scan(seq=sequence, lowscore=1)
>>> result = ScanProsite.read(handle)
>>> result.n_match
2
```

Chapter 14

Going 3D: The PDB module

Bio.PDB is a Biopython module that focuses on working with crystal structures of biological macromolecules. Among other things, Bio.PDB includes a PDBParser class that produces a Structure object, which can be used to access the atomic data in the file in a convenient manner. There is limited support for parsing the information contained in the PDB header. PDB file format is no longer being modified or extended to support new content and PDBx/mmCIF became the standard PDB archive format in 2014. All the Worldwide Protein Data Bank (wwPDB) sites uses the macromolecular Crystallographic Information File (mmCIF) data dictionaries to describe the information content of PDB entries. mmCIF uses a flexible and extensible key-value pair format for representing macromolecular structural data and imposes no limitations for the number of atoms, residues or chains that can be represented in a single PDB entry (no split entries!).

14.1 Reading and writing crystal structure files

14.1.1 Reading an mmCIF file

First create an `MMCIFParser` object:

```
>>> from Bio.PDB.MMCIFParser import MMCIFParser
>>> parser = MMCIFParser()
```

Then use this parser to create a structure object from the mmCIF file:

```
>>> structure = parser.get_structure("1fat", "1fat.cif")
```

To have some more low level access to an mmCIF file, you can use the `MMCIF2Dict` class to create a Python dictionary that maps all mmCIF tags in an mmCIF file to their values. Whether there are multiple values (like in the case of tag `_atom_site.Cartn_y`, which holds the y coordinates of all atoms) or a single value (like the initial deposition date), the tag is mapped to a list of values. The dictionary is created from the mmCIF file as follows:

```
>>> from Bio.PDB.MMCIF2Dict import MMCIF2Dict
>>> mmcif_dict = MMCIF2Dict("1FAT.cif")
```

Example: get the solvent content from an mmCIF file:

```
>>> sc = mmcif_dict["_exptl_crystal.density_percent_sol"]
```

Example: get the list of the y coordinates of all atoms

```
>>> y_list = mmcif_dict["_atom_site.Cartn_y"]
```

14.1.2 Reading files in the MMTF format

You can use the direct MMTFParser to read a structure from a file:

```
>>> from Bio.PDB.mmtf import MMTFParser
>>> structure = MMTFParser.get_structure("PDB/4CUP.mmtf")
```

Or you can use the same class to get a structure by its PDB ID:

```
>>> structure = MMTFParser.get_structure_from_url("4CUP")
```

This gives you a Structure object as if read from a PDB or mmCIF file.

You can also have access to the underlying data using the external MMTF library which Biopython is using internally:

```
>>> from mmtf import fetch
>>> decoded_data = fetch("4CUP")
```

For example you can access just the X-coordinate.

```
>>> print(decoded_data.x_coord_list)
```

14.1.3 Reading a PDB file

First we create a `PDBParser` object:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser(PERMISSIVE=1)
```

The `PERMISSIVE` flag indicates that a number of common problems (see 14.7.1) associated with PDB files will be ignored (but note that some atoms and/or residues will be missing). If the flag is not present a `PDBConstructionException` will be generated if any problems are detected during the parse operation.

The Structure object is then produced by letting the `PDBParser` object parse a PDB file (the PDB file in this case is called `pdb1fat.ent`, `1fat` is a user defined name for the structure):

```
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename)
```

You can extract the header and trailer (simple lists of strings) of the PDB file from the PDBParser object with the `get_header` and `get_trailer` methods. Note however that many PDB files contain headers with incomplete or erroneous information. Many of the errors have been fixed in the equivalent mmCIF files. *Hence, if you are interested in the header information, it is a good idea to extract information from mmCIF files using the `MMCIF2Dict` tool described above, instead of parsing the PDB header.*

Now that is clarified, let's return to parsing the PDB header. The structure object has an attribute called `header` which is a Python dictionary that maps header records to their values.

Example:

```
>>> resolution = structure.header["resolution"]
>>> keywords = structure.header["keywords"]
```

The available keys are `name`, `head`, `deposition_date`, `release_date`, `structure_method`, `resolution`, `structure_reference` (which maps to a list of references), `journal_reference`, `author`, `compound` (which maps to a dictionary with various information about the crystallized compound), `has_missing_residues`, `missing_residues`, and `astral` (which maps to dictionary with additional information about the domain if present).

`has_missing_residues` maps to a bool that is True if at least one non-empty `REMARK 465` header line was found. In this case you should assume that the molecule used in the experiment has some residues for which no ATOM coordinates could be determined. `missing_residues` maps to a list of dictionaries with information about the missing residues. *The list of missing residues will be empty or incomplete if the PDB header does not follow the template from the PDB specification.*

The dictionary can also be created without creating a `Structure` object, ie. directly from the PDB file:

```
>>> from Bio.PDB import parse_pdb_header
>>> with open(filename, "r") as handle:
...     header_dict = parse_pdb_header(handle)
...
```

14.1.4 Reading a PQR file

In order to parse a PQR file, proceed in a similar manner as in the case of PDB files:

Create a `PDBParser` object, using the `is_pqr` flag:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> pqr_parser = PDBParser(PERMISSIVE=1, is_pqr=True)
```

The `is_pqr` flag set to `True` indicates that the file to be parsed is a PQR file, and that the parser should read the atomic charge and radius fields for each atom entry. Following the same procedure as for PQR files, a Structure object is then produced, and the PQR file is parsed.

```
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> structure = parser.get_structure(structure_id, filename, is_pqr=True)
```

14.1.5 Reading files in the PDB XML format

That's not yet supported, but we are definitely planning to support that in the future (it's not a lot of work). Contact the Biopython developers via the mailing list if you need this.

14.1.6 Writing mmCIF files

The `MMCIFIO` class can be used to write structures to the mmCIF file format:

```
>>> io = MMCIFIO()
>>> io.set_structure(s)
>>> io.save("out.cif")
```

The `Select` class can be used in a similar way to `PDBIO` below. mmCIF dictionaries read using `MMCIF2Dict` can also be written:

```
>>> io = MMCIFIO()
>>> io.set_dict(d)
>>> io.save("out.cif")
```

14.1.7 Writing PDB files

Use the `PDBIO` class for this. It's easy to write out specific parts of a structure too, of course.

Example: saving a structure

```
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("out.pdb")
```

If you want to write out a part of the structure, make use of the `Select` class (also in `PDBIO`). Select has four methods:

- `accept_model(model)`
- `accept_chain(chain)`
- `accept_residue(residue)`
- `accept_atom(atom)`

By default, every method returns 1 (which means the model/chain/residue/atom is included in the output). By subclassing `Select` and returning 0 when appropriate you can exclude models, chains, etc. from the output. Cumbersome maybe, but very powerful. The following code only writes out glycine residues:

```
>>> class GlySelect(Select):
...     def accept_residue(self, residue):
...         if residue.get_name() == "GLY":
...             return True
...         else:
...             return False
...
>>> io = PDBIO()
>>> io.set_structure(s)
>>> io.save("gly_only.pdb", GlySelect())
```

If this is all too complicated for you, the `Dice` module contains a handy `extract` function that writes out all residues in a chain between a start and end residue.

14.1.8 Writing PQR files

Use the `PDBIO` class as you would for a PDB file, with the flag `is_pqr=True`. The PDBIO methods can be used in the case of PQR files as well.

Example: writing a PQR file

```
>>> io = PDBIO(is_pqr=True)
>>> io.set_structure(s)
>>> io.save("out.pdb")
```

14.1.9 Writing MMTF files

To write structures to the MMTF file format:

```
>>> from Bio.PDB.mmtf import MMTFIO
>>> io = MMTFIO()
>>> io.set_structure(s)
>>> io.save("out.mmtf")
```

The `Select` class can be used as above. Note that the bonding information, secondary structure assignment and some other information contained in standard MMTF files is not written out as it is not easy to determine from the structure object. In addition, molecules that are grouped into the same entity in standard MMTF files are treated as separate entities by `MMTFIO`.

14.2 Structure representation

The overall layout of a `Structure` object follows the so-called SMCRA (Structure/Model/Chain/Residue/Atom) architecture:

- A structure consists of models
- A model consists of chains
- A chain consists of residues
- A residue consists of atoms

This is the way many structural biologists/bioinformaticians think about structure, and provides a simple but efficient way to deal with structure. Additional stuff is essentially added when needed. A UML diagram of the `Structure` object (forget about the `Disordered` classes for now) is shown in Figure 14.1. Such a data structure is not necessarily best suited for the representation of the macromolecular content of a structure, but it is absolutely necessary for a good interpretation of the data present in a file that describes the structure (typically a PDB or MMCIF file). If this hierarchy cannot represent the contents of a structure file, it is fairly certain that the file contains an error or at least does not describe the structure unambiguously. If a SMCRA data structure cannot be generated, there is reason to suspect a problem. Parsing a PDB file can thus be used to detect likely problems. We will give several examples of this in section 14.7.1.

Structure, Model, Chain and Residue are all subclasses of the Entity base class. The Atom class only (partly) implements the Entity interface (because an Atom does not have children).

For each Entity subclass, you can extract a child by using a unique id for that child as a key (e.g. you can extract an Atom object from a Residue object by using an atom name string as a key, you can extract a Chain object from a Model object by using its chain identifier as a key).

Disordered atoms and residues are represented by DisorderedAtom and DisorderedResidue classes, which are both subclasses of the DisorderedEntityWrapper base class. They hide the complexity associated with disorder and behave exactly as Atom and Residue objects.

In general, a child Entity object (i.e. Atom, Residue, Chain, Model) can be extracted from its parent (i.e. Residue, Chain, Model, Structure, respectively) by using an id as a key.

```
>>> child_entity = parent_entity[child_id]
```

You can also get a list of all child Entities of a parent Entity object. Note that this list is sorted in a specific way (e.g. according to chain identifier for Chain objects in a Model object).

```
>>> child_list = parent_entity.get_list()
```

You can also get the parent from a child:

```
>>> parent_entity = child_entity.get_parent()
```

At all levels of the SMCRA hierarchy, you can also extract a *full id*. The full id is a tuple containing all id's starting from the top object (Structure) down to the current object. A full id for a Residue object e.g. is something like:

```
>>> full_id = residue.get_full_id()
>>> print(full_id)
("1abc", 0, "A", ("", 10, "A"))
```

This corresponds to:

- The Structure with id `"1abc"`

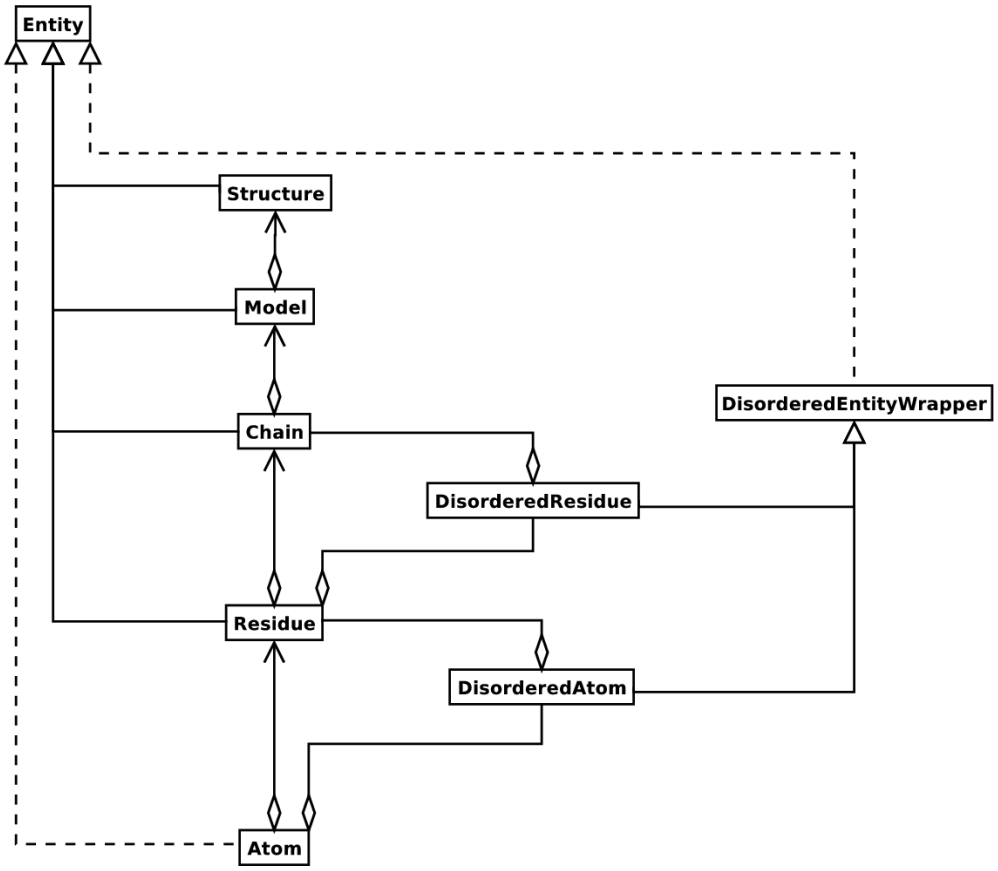


Figure 14.1: UML diagram of SMCRA architecture of the `Structure` class used to represent a macromolecular structure. Full lines with diamonds denote aggregation, full lines with arrows denote referencing, full lines with triangles denote inheritance and dashed lines with triangles denote interface realization.

- The Model with id `0`
- The Chain with id `"A"`
- The Residue with id `("", 10, "A")`

The Residue id indicates that the residue is not a hetero-residue (nor a water) because it has a blank hetero field, that its sequence identifier is 10 and that its insertion code is `"A"`.

To get the entity's id, use the `get_id` method:

```
>>> entity.get_id()
```

You can check if the entity has a child with a given id by using the `has_id` method:

```
>>> entity.has_id(entity_id)
```

The length of an entity is equal to its number of children:

```
>>> nr_children = len(entity)
```

It is possible to delete, rename, add, etc. child entities from a parent entity, but this does not include any sanity checks (e.g. it is possible to add two residues with the same id to one chain). This really should be done via a nice Decorator class that includes integrity checking, but you can take a look at the code (Entity.py) if you want to use the raw interface.

14.2.1 Structure

The Structure object is at the top of the hierarchy. Its id is a user given string. The Structure contains a number of Model children. Most crystal structures (but not all) contain a single model, while NMR structures typically consist of several models. Disorder in crystal structures of large parts of molecules can also result in several models.

14.2.2 Model

The id of the Model object is an integer, which is derived from the position of the model in the parsed file (they are automatically numbered starting from 0). Crystal structures generally have only one model (with id 0), while NMR files usually have several models. Whereas many PDB parsers assume that there is only one model, the `Structure` class in `Bio.PDB` is designed such that it can easily handle PDB files with more than one model.

As an example, to get the first model from a Structure object, use

```
>>> first_model = structure[0]
```

The Model object stores a list of Chain children.

14.2.3 Chain

The id of a Chain object is derived from the chain identifier in the PDB/mmCIF file, and is a single character (typically a letter). Each Chain in a Model object has a unique id. As an example, to get the Chain object with identifier "A" from a Model object, use

```
>>> chain_A = model["A"]
```

The Chain object stores a list of Residue children.

14.2.4 Residue

A residue id is a tuple with three elements:

- The **hetero-field** (hetfield): this is
  - `'W'` in the case of a water molecule;
  - `'H_'` followed by the residue name for other hetero residues (e.g. `'H_GLC'` in the case of a glucose molecule);
  - blank for standard amino and nucleic acids.

  This scheme is adopted for reasons described in section 14.4.1.
- The **sequence identifier** (resseq), an integer describing the position of the residue in the chain (e.g., 100);
- The **insertion code** (icode); a string, e.g. 'A'. The insertion code is sometimes used to preserve a certain desirable residue numbering scheme. A Ser 80 insertion mutant (inserted e.g. between a Thr 80 and an Asn 81 residue) could e.g. have sequence identifiers and insertion codes as follows: Thr 80 A, Ser 80 B, Asn 81. In this way the residue numbering scheme stays in tune with that of the wild type structure.

The id of the above glucose residue would thus be `('H_GLC', 100, 'A')`. If the hetero-flag and insertion code are blank, the sequence identifier alone can be used:

```
# Full id
>>> residue = chain[(" ", 100, " ")]
# Shortcut id
>>> residue = chain[100]
```

The reason for the hetero-flag is that many, many PDB files use the same sequence identifier for an amino acid and a hetero-residue or a water, which would create obvious problems if the hetero-flag was not used.

Unsurprisingly, a Residue object stores a set of Atom children. It also contains a string that specifies the residue name (e.g. "ASN") and the segment identifier of the residue (well known to X-PLOR users, but not used in the construction of the SMCRA data structure).

Let's look at some examples. Asn 10 with a blank insertion code would have residue id `(' ', 10, ' ')`. Water 10 would have residue id `('W', 10, ' ')`. A glucose molecule (a hetero residue with residue name GLC) with sequence identifier 10 would have residue id `('H_GLC', 10, ' ')`. In this way, the three residues (with the same insertion code and sequence identifier) can be part of the same chain because their residue id's are distinct.

In most cases, the hetflag and insertion code fields will be blank, e.g. `(' ', 10, ' ')`. In these cases, the sequence identifier can be used as a shortcut for the full id:

```
# use full id
>>> res10 = chain[(" ", 10, " ")]
# use shortcut
>>> res10 = chain[10]
```

Each Residue object in a Chain object should have a unique id. However, disordered residues are dealt with in a special way, as described in section 14.3.3.

A Residue object has a number of additional methods:

```
>>> residue.get_resname()  # returns the residue name, e.g. "ASN"
>>> residue.is_disordered()  # returns 1 if the residue has disordered atoms
>>> residue.get_segid()  # returns the SEGID, e.g. "CHN1"
>>> residue.has_id(name)  # test if a residue has a certain atom
```

You can use `is_aa(residue)` to test if a Residue object is an amino acid.

14.2.5 Atom

The Atom object stores the data associated with an atom, and has no children. The id of an atom is its atom name (e.g. "OG" for the side chain oxygen of a Ser residue). An Atom id needs to be unique in a Residue. Again, an exception is made for disordered atoms, as described in section 14.3.2.

The atom id is simply the atom name (eg. `'CA'`). In practice, the atom name is created by stripping all spaces from the atom name in the PDB file.

However, in PDB files, a space can be part of an atom name. Often, calcium atoms are called `'CA..'` in order to distinguish them from Cα atoms (which are called `'.CA.'`). In cases were stripping the spaces would create problems (ie. two atoms called `'CA'` in the same residue) the spaces are kept.

In a PDB file, an atom name consists of 4 chars, typically with leading and trailing spaces. Often these spaces can be removed for ease of use (e.g. an amino acid Cα atom is labeled ".CA." in a PDB file, where the dots represent spaces). To generate an atom name (and thus an atom id) the spaces are removed, unless this would result in a name collision in a Residue (i.e. two Atom objects with the same atom name and id). In the latter case, the atom name including spaces is tried. This situation can e.g. happen when one residue contains atoms with names ".CA." and "CA..", although this is not very likely.

The atomic data stored includes the atom name, the atomic coordinates (including standard deviation if present), the B factor (including anisotropic B factors and standard deviation if present), the altloc specifier and the full atom name including spaces. Less used items like the atom element number or the atomic charge sometimes specified in a PDB file are not stored.

To manipulate the atomic coordinates, use the `transform` method of the `Atom` object. Use the `set_coord` method to specify the atomic coordinates directly.

An Atom object has the following additional methods:

```
>>> a.get_name()  # atom name (spaces stripped, e.g. "CA")
>>> a.get_id()  # id (equals atom name)
>>> a.get_coord()  # atomic coordinates
>>> a.get_vector()  # atomic coordinates as Vector object
>>> a.get_bfactor()  # isotropic B factor
>>> a.get_occupancy()  # occupancy
>>> a.get_altloc()  # alternative location specifier
>>> a.get_sigatm()  # standard deviation of atomic parameters
>>> a.get_siguij()  # standard deviation of anisotropic B factor
>>> a.get_anisou()  # anisotropic B factor
>>> a.get_fullname()  # atom name (with spaces, e.g. ".CA.")
```

To represent the atom coordinates, siguij, anisotropic B factor and sigatm Numpy arrays are used.

The `get_vector` method returns a `Vector` object representation of the coordinates of the `Atom` object, allowing you to do vector operations on atomic coordinates. `Vector` implements the full set of 3D vector operations, matrix multiplication (left and right) and some advanced rotation-related operations as well.

As an example of the capabilities of Bio.PDB's `Vector` module, suppose that you would like to find the position of a Gly residue's Cβ atom, if it had one. Rotating the N atom of the Gly residue along the Cα-C bond over -120 degrees roughly puts it in the position of a virtual Cβ atom. Here's how to do it, making use of the `rotaxis` method (which can be used to construct a rotation around a certain axis) of the `Vector` module:

```
# get atom coordinates as vectors
>>> n = residue["N"].get_vector()
>>> c = residue["C"].get_vector()
>>> ca = residue["CA"].get_vector()
# center at origin
>>> n = n - ca
>>> c = c - ca
```

```
# find rotation matrix that rotates n
# -120 degrees along the ca-c vector
>>> rot = rotaxis(-pi * 120.0 / 180.0, c)
# apply rotation to ca-n vector
>>> cb_at_origin = n.left_multiply(rot)
# put on top of ca atom
>>> cb = cb_at_origin + ca
```

This example shows that it's possible to do some quite nontrivial vector operations on atomic data, which can be quite useful. In addition to all the usual vector operations (cross (use `**`), and dot (use `*`) product, angle, norm, etc.) and the above mentioned `rotaxis` function, the `Vector` module also has methods to rotate (`rotmat`) or reflect (`refmat`) one vector on top of another.

14.2.6 Extracting a specific `Atom/Residue/Chain/Model` from a Structure

These are some examples:

```
>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[100]
>>> atom = residue["CA"]
```

Note that you can use a shortcut:

```
>>> atom = structure[0]["A"][100]["CA"]
```

14.3 Disorder

Bio.PDB can handle both disordered atoms and point mutations (i.e. a Gly and an Ala residue in the same position).

14.3.1 General approach

Disorder should be dealt with from two points of view: the atom and the residue points of view. In general, we have tried to encapsulate all the complexity that arises from disorder. If you just want to loop over all Cα atoms, you do not care that some residues have a disordered side chain. On the other hand it should also be possible to represent disorder completely in the data structure. Therefore, disordered atoms or residues are stored in special objects that behave as if there is no disorder. This is done by only representing a subset of the disordered atoms or residues. Which subset is picked (e.g. which of the two disordered OG side chain atom positions of a Ser residue is used) can be specified by the user.

14.3.2 Disordered atoms

Disordered atoms are represented by ordinary `Atom` objects, but all `Atom` objects that represent the same physical atom are stored in a `DisorderedAtom` object (see Figure 14.1). Each `Atom` object in a `DisorderedAtom` object can be uniquely indexed using its altloc specifier. The `DisorderedAtom` object forwards all uncaught method calls to the selected Atom object, by default the one that represents the atom with the highest occupancy. The user can of course change the selected `Atom` object, making use of its altloc specifier. In this way atom disorder is represented correctly without much additional complexity. In other words, if you are not interested in atom disorder, you will not be bothered by it.

Each disordered atom has a characteristic altloc identifier. You can specify that a `DisorderedAtom` object should behave like the `Atom` object associated with a specific altloc identifier:

```
>>> atom.disordered_select("A")  # select altloc A atom
>>> print(atom.get_altloc())
"A"
>>> atom.disordered_select("B")  # select altloc B atom
>>> print(atom.get_altloc())
"B"
```

14.3.3 Disordered residues

Common case

The most common case is a residue that contains one or more disordered atoms. This is evidently solved by using DisorderedAtom objects to represent the disordered atoms, and storing the DisorderedAtom object in a Residue object just like ordinary Atom objects. The DisorderedAtom will behave exactly like an ordinary atom (in fact the atom with the highest occupancy) by forwarding all uncaught method calls to one of the Atom objects (the selected Atom object) it contains.

Point mutations

A special case arises when disorder is due to a point mutation, i.e. when two or more point mutants of a polypeptide are present in the crystal. An example of this can be found in PDB structure 1EN2.

Since these residues belong to a different residue type (e.g. let's say Ser 60 and Cys 60) they should not be stored in a single `Residue` object as in the common case. In this case, each residue is represented by one `Residue` object, and both `Residue` objects are stored in a single `DisorderedResidue` object (see Figure 14.1).

The `DisorderedResidue` object forwards all uncaught methods to the selected `Residue` object (by default the last `Residue` object added), and thus behaves like an ordinary residue. Each `Residue` object in a `DisorderedResidue` object can be uniquely identified by its residue name. In the above example, residue Ser 60 would have id "SER" in the `DisorderedResidue` object, while residue Cys 60 would have id "CYS". The user can select the active `Residue` object in a `DisorderedResidue` object via this id.

Example: suppose that a chain has a point mutation at position 10, consisting of a Ser and a Cys residue. Make sure that residue 10 of this chain behaves as the Cys residue.

```
>>> residue = chain[10]
>>> residue.disordered_select("CYS")
```

In addition, you can get a list of all `Atom` objects (ie. all `DisorderedAtom` objects are 'unpacked' to their individual `Atom` objects) using the `get_unpacked_list` method of a `(Disordered)Residue` object.

14.4 Hetero residues

14.4.1 Associated problems

A common problem with hetero residues is that several hetero and non-hetero residues present in the same chain share the same sequence identifier (and insertion code). Therefore, to generate a unique id for each hetero residue, waters and other hetero residues are treated in a different way.

Remember that Residue object have the tuple (hetfield, resseq, icode) as id. The hetfield is blank (" ") for amino and nucleic acids, and a string for waters and other hetero residues. The content of the hetfield is explained below.

14.4.2 Water residues

The hetfield string of a water residue consists of the letter "W". So a typical residue id for a water is ("W", 1, " ").

14.4.3 Other hetero residues

The hetfield string for other hetero residues starts with "H_" followed by the residue name. A glucose molecule e.g. with residue name "GLC" would have hetfield "H_GLC". Its residue id could e.g. be ("H_GLC", 1, " ").

14.5 Navigating through a Structure object

Parse a PDB file, and extract some Model, Chain, Residue and Atom objects

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> parser = PDBParser()
>>> structure = parser.get_structure("test", "1fat.pdb")
>>> model = structure[0]
>>> chain = model["A"]
>>> residue = chain[1]
>>> atom = residue["CA"]
```

Iterating through all atoms of a structure

```
>>> p = PDBParser()
>>> structure = p.get_structure("X", "pdb1fat.ent")
>>> for model in structure:
...     for chain in model:
...         for residue in chain:
...             for atom in residue:
...                 print(atom)
...
```

There is a shortcut if you want to iterate over all atoms in a structure:

```
>>> atoms = structure.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Similarly, to iterate over all atoms in a chain, use

```
>>> atoms = chain.get_atoms()
>>> for atom in atoms:
...     print(atom)
...
```

Iterating over all residues of a model

or if you want to iterate over all residues in a model:

```
>>> residues = model.get_residues()
>>> for residue in residues:
...     print(residue)
...
```

You can also use the `Selection.unfold_entities` function to get all residues from a structure:

```
>>> res_list = Selection.unfold_entities(structure, "R")
```

or to get all atoms from a chain:

```
>>> atom_list = Selection.unfold_entities(chain, "A")
```

Obviously, `A=atom, R=residue, C=chain, M=model, S=structure`. You can use this to go up in the hierarchy, e.g. to get a list of (unique) `Residue` or `Chain` parents from a list of `Atoms`:

```
>>> residue_list = Selection.unfold_entities(atom_list, "R")
>>> chain_list = Selection.unfold_entities(atom_list, "C")
```

For more info, see the API documentation.

Extract hetero residue from chain (e.g. glucose (GLC) moiety with resseq 10)

```
>>> residue_id = ("H_GLC", 10, " ")
>>> residue = chain[residue_id]
```

Print all hetero residues in chain

```
>>> for residue in chain.get_list():
...     residue_id = residue.get_id()
...     hetfield = residue_id[0]
...     if hetfield[0] == "H":
...         print(residue_id)
...
```

Print out coordinates of all CA atoms in structure with B factor over 50

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.has_id("CA"):
...                 ca = residue["CA"]
...                 if ca.get_bfactor() > 50.0:
...                     print(ca.get_coord())
...
```

Print out all the residues that contain disordered atoms

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.is_disordered():
...                 resseq = residue.get_id()[1]
...                 resname = residue.get_resname()
...                 model_id = model.get_id()
...                 chain_id = chain.get_id()
...                 print(model_id, chain_id, resname, resseq)
...
```

Loop over all disordered atoms, and select all atoms with altloc A (if present)

This will make sure that the SMCRA data structure will behave as if only the atoms with altloc A are present.

```
>>> for model in structure.get_list():
...     for chain in model.get_list():
...         for residue in chain.get_list():
...             if residue.is_disordered():
...                 for atom in residue.get_list():
...                     if atom.is_disordered():
...                         if atom.disordered_has_id("A"):
...                             atom.disordered_select("A")
...
```

Extracting polypeptides from a `Structure` object

To extract polypeptides from a structure, construct a list of `Polypeptide` objects from a `Structure` object using `PolypeptideBuilder` as follows:

```
>>> model_nr = 1
>>> polypeptide_list = build_peptides(structure, model_nr)
>>> for polypeptide in polypeptide_list:
...     print(polypeptide)
...
```

A Polypeptide object is simply a UserList of Residue objects, and is always created from a single Model (in this case model 1). You can use the resulting `Polypeptide` object to get the sequence as a `Seq` object or to get a list of Cα atoms as well. Polypeptides can be built using a C-N or a Cα-Cα distance criterion.

Example:

```
# Using C-N
>>> ppb = PPBuilder()
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
...
# Using CA-CA
>>> ppb = CaPPBuilder()
>>> for pp in ppb.build_peptides(structure):
...     print(pp.get_sequence())
...
```

Note that in the above case only model 0 of the structure is considered by `PolypeptideBuilder`. However, it is possible to use `PolypeptideBuilder` to build `Polypeptide` objects from `Model` and `Chain` objects as well.

Obtaining the sequence of a structure

The first thing to do is to extract all polypeptides from the structure (as above). The sequence of each polypeptide can then easily be obtained from the `Polypeptide` objects. The sequence is represented as a Biopython `Seq` object.

Example:

```
>>> seq = polypeptide.get_sequence()
>>> seq
Seq('SNDIYFNFQRFNETNLILQRDASVSSSGQLRLTNLN')
```

14.6 Analyzing structures

14.6.1 Measuring distances

The minus operator for atoms has been overloaded to return the distance between two atoms.

```
# Get some atoms
>>> ca1 = residue1["CA"]
>>> ca2 = residue2["CA"]
# Simply subtract the atoms to get their distance
>>> distance = ca1 - ca2
```

14.6.2 Measuring angles

Use the vector representation of the atomic coordinates, and the `calc_angle` function from the `Vector` module:

```
>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> angle = calc_angle(vector1, vector2, vector3)
```

14.6.3 Measuring torsion angles

Use the vector representation of the atomic coordinates, and the `calc_dihedral` function from the `Vector` module:

```
>>> vector1 = atom1.get_vector()
>>> vector2 = atom2.get_vector()
>>> vector3 = atom3.get_vector()
>>> vector4 = atom4.get_vector()
>>> angle = calc_dihedral(vector1, vector2, vector3, vector4)
```

14.6.4 Internal coordinates - distances, angles, torsion angles, distance plots, etc

Protein structures are normally supplied in 3D XYZ coordinates relative to a fixed origin, as in a PDB or mmCIF file. The `internal_coords` module facilitates converting this system to and from bond lengths, angles and dihedral angles. In addition to supporting standard *psi, phi, chi*, etc. calculations on protein structures, this representation is invariant to translation and rotation, and the implementation exposes multiple benefits for structure analysis.

First load up some modules here for later examples:

```
>>> from Bio.PDB.PDBParser import PDBParser
>>> from Bio.PDB.Chain import Chain
>>> from Bio.PDB.internal_coords import *
>>> from Bio.PDB.PICIO import write_PIC, read_PIC, read_PIC_seq
>>> from Bio.PDB.ic_rebuild import write_PDB, IC_duplicate, structure_rebuild_test
>>> from Bio.PDB.SCADIO import write_SCAD
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.PDB.PDBIO import PDBIO
>>> import numpy as np
```

14.6.4.1 Accessing dihedrals, angles and bond lengths

We start with the simple case of computing internal coordinates for a structure:

```
>>> # load a structure as normal, get first chain
>>> parser = PDBParser()
>>> myProtein = parser.get_structure("1a8o", "1A8O.pdb")
>>> myChain = myProtein[0]["A"]

>>> # compute bond lengths, angles, dihedral angles
>>> myChain.atom_to_internal_coordinates(verbose=True)
chain break at THR  186  due to MaxPeptideBond (1.4 angstroms) exceeded
chain break at THR  216  due to MaxPeptideBond (1.4 angstroms) exceeded
```

The chain break warnings for 1A8O are suppressed by removing the `verbose=True` option above. To avoid the creation of a break and instead allow unrealistically long N-C bonds, override the class variable `MaxPeptideBond`, e.g.:

```
>>> IC_Chain.MaxPeptideBond = 4.0
>>> myChain.internal_coord = None  # force re-loading structure data with new cutoff
>>> myChain.atom_to_internal_coordinates(verbose=True)
```

At this point the values are available at both the chain and residue level. The first residue of 1A8O is HETATM MSE (selenomethionine), so we investigate residue 2 below using either canonical names or atom specifiers. Here we obtain the *chi1* dihedral and *tau* angles by name and by atom sequence, and the Cα-Cβ distance by specifying the atom pair:

```
>>> r2 = myChain.child_list[1]
>>> r2
<Residue ASP het=  resseq=152 icode= >
>>> r2ic = r2.internal_coord
>>> print(r2ic, ":", r2ic.pretty_str(), ":", r2ic.rbase, ":", r2ic.lc)
('1a8o', 0, 'A', (' ', 152, ' ')) : ASP  152  : (152, None, 'D') : D

>>> r2chi1 = r2ic.get_angle("chi1")
>>> print(round(r2chi1, 2))
-144.86
>>> r2ic.get_angle("chi1") == r2ic.get_angle("N:CA:CB:CG")
True
>>> print(round(r2ic.get_angle("tau"), 2))
113.45
>>> r2ic.get_angle("tau") == r2ic.get_angle("N:CA:C")
True
>>> print(round(r2ic.get_length("CA:CB"), 2))
1.53
```

The `Chain.internal_coord` object holds arrays and dictionaries of hedra (3 bonded atoms) and dihedra (4 bonded atoms) objects. The dictionaries are indexed by tuples of `AtomKey` objects; `AtomKey` objects capture residue position, insertion code, 1 or 3-character residue name, atom name, altloc and occupancy.

Below we obtain the same *chi1* and *tau* angles as above by indexing the `Chain` arrays directly, using `AtomKeys` to index the `Chain` arrays:

```
>>> myCic = myChain.internal_coord
```

```
>>> r2chi1_object = r2ic.pick_angle("chi1")
>>> # or same thing (as for get_angle() above):
>>> r2chi1_object == r2ic.pick_angle("N:CA:CB:CG")
True
>>> r2chi1_key = r2chi1_object.atomkeys
>>> r2chi1_key  # r2chi1_key is tuple of AtomKeys
(152_D_N, 152_D_CA, 152_D_CB, 152_D_CG)

>>> r2chi1_index = myCic.dihedraNdx[r2chi1_key]
>>> # or same thing:
>>> r2chi1_index == r2chi1_object.ndx
True
>>> print(round(myCic.dihedraAngle[r2chi1_index], 2))
-144.86
>>> # also:
>>> r2chi1_object == myCic.dihedra[r2chi1_key]
True

>>> # hedra angles are similar:
>>> r2tau = r2ic.pick_angle("tau")
>>> print(round(myCic.hedraAngle[r2tau.ndx], 2))
113.45
```

Obtaining bond length data at the `Chain` level is more complicated (and not recommended). As shown here, multiple hedra will share a single bond in different positions:

```
>>> r2CaCb = r2ic.pick_length("CA:CB")  # returns list of hedra containing bond
>>> r2CaCb[0][0].atomkeys
(152_D_CB, 152_D_CA, 152_D_C)
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2))  # position 1-2
1.53
>>> r2CaCb[0][1].atomkeys
(152_D_N, 152_D_CA, 152_D_CB)
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2))  # position 2-3
1.53
>>> r2CaCb[0][2].atomkeys
(152_D_CA, 152_D_CB, 152_D_CG)
>>> print(round(myCic.hedraL12[r2CaCb[0][2].ndx], 2))  # position 1-2
1.53
```

Please use the `Residue` level `set_length` function instead.

14.6.4.2 Testing structures for completeness

Missing atoms and other issues can cause problems when rebuilding a structure. Use `structure_rebuild_test` to determine quickly if a structure has sufficient data for a clean rebuild. Add `verbose=True` and/or inspect the result dictionary for more detail:

```
>>> # check myChain makes sense (can get angles and rebuild same structure)
>>> resultDict = structure_rebuild_test(myChain)
>>> resultDict["pass"]
True
```

14.6.4.3 Modifying and rebuilding structures

It's preferable to use the residue level `set_angle` and `set_length` facilities for modifying internal coordinates rather than directly accessing the `Chain` structures. While directly modifying hedra angles is safe, bond lengths appear in multiple overlapping hedra as noted above, and this is handled by `set_length`. When applied to a dihedral angle, `set_angle` will wrap the result to +/-180 and rotate adjacent dihedra as well (such as both bonds for an isoleucine *chi1* angle - which is probably what you want).

```
>>> # rotate residue 2 chi1 angle by -120 degrees
>>> r2ic.set_angle("chi1", r2chi1 - 120.0)
>>> print(round(r2ic.get_angle("chi1"), 2))
95.14
>>> r2ic.set_length("CA:CB", 1.49)
>>> print(round(myCic.hedraL12[r2CaCb[0][0].ndx], 2))  # Cb-Ca-C position 1-2
1.49
```

Rebuilding a structure from internal coordinates is a simple call to `internal_to_atom_coordinates()`:

```
>>> myChain.internal_to_atom_coordinates()

>>> # just for proof:
>>> myChain.internal_coord = None  # all internal_coord data removed, only atoms left
>>> myChain.atom_to_internal_coordinates()  # re-generate internal coordinates
>>> r2ic = myChain.child_list[1].internal_coord
>>> print(round(r2ic.get_angle("chi1"), 2))  # show measured values match what was set above
95.14
>>> print(round(myCic.hedraL23[r2CaCb[0][1].ndx], 2))  # N-Ca-Cb position 2-3
1.49
```

The generated structure can be written with PDBIO, as normal:

```
write_PDB(myProtein, "myChain.pdb")
# or just the ATOM records without headers:
io = PDBIO()
io.set_structure(myProtein)
io.save("myChain2.pdb")
```

14.6.4.4 Protein Internal Coordinate (.pic) files and default values

A file format is defined in the `PICIO` module to describe protein chains as hedra and dihedra relative to initial coordinates. All parts of the file other than the residue sequence information (e.g. `('1A8O', 0, 'A', (' ', 153, ' ')) ILE`) are optional, and will be filled in with default values if not specified and `read_PIC` is called with the `defaults=True` option. Default values are calculated from Sep 2019 Dunbrack cullpdb_pc20_res2.2_R1.0.

Here we write 'myChain' as a `.pic` file of internal coordinate specifications and then read it back in as 'myProtein2'.

```
# write chain as 'protein internal coordinates' (.pic) file
write_PIC(myProtein, "myChain.pic")
# read .pic file
myProtein2 = read_PIC("myChain.pic")
```

As all internal coordinate values can be replaced with defaults, `PICIO.read_PIC_seq` is supplied as a utility function to create a valid (mostly helical) default structure from an input sequence:

```
# create default structure for random sequence by reading as .pic file
myProtein3 = read_PIC_seq(
    SeqRecord(
        Seq("GAVLIMFPSTCNQYWDEHKR"),
        id="1RND",
        description="my random sequence",
    )
)
myProtein3.internal_to_atom_coordinates()
write_PDB(myProtein3, "myRandom.pdb")
```

It may be of interest to explore the accuracy required in e.g. *omega* angles (180.0), hedra angles and/or bond lengths when generating structures from internal coordinates. The picFlags option to `write_PIC` enables this, allowing the selection of data to be written to the .pic file vs. left unspecified to get default values.

Various combinations are possible and some presets are supplied, for example `classic` will write only *psi, phi, tau*, proline *omega* and sidechain *chi* angles to the .pic file:

```
write_PIC(myProtein, "myChain.pic", picFlags=IC_Residue.pic_flags.classic)
myProtein2 = read_PIC("myChain.pic", defaults=True)
```

14.6.4.5 Accessing the all-atom AtomArray

All 3D XYZ coordinates in Biopython `Atom` objects are moved to a single large array in the `Chain` class and replaced by Numpy 'views' into this array in an early step of `atom_to_internal_coordinates`. Software accessing Biopython `Atom` coordinates is not affected, but the new array may offer efficiencies for future work.

Unlike the `Atom` XYZ coordinates, `AtomArray` coordinates are homogeneous, meaning they are arrays like `[ x y z 1.0]` with 1.0 as the fourth element. This facilitates efficient transformation using combined translation and rotation matrices throughout the `internal_coords` module. There is a corresponding `AtomArrayIndex` dictionary, mapping `AtomKeys` to their coordinates.

Here we demonstrate reading coordinates for a specific Cβ atom from the array, then show that modifying the array value modifies the `Atom` object at the same time:

```
>>> # access the array of all atoms for the chain, e.g. r2 above is residue 152 C-beta
>>> r2_cBeta_index = myChain.internal_coord.atomArrayIndex[AtomKey("152_D_CB")]
>>> r2_cBeta_coords = myChain.internal_coord.atomArray[r2_cBeta_index]
>>> print(np.round(r2_cBeta_coords, 2))
[-0.75 -1.18 -0.51  1.  ]

>>> # the Biopython Atom coord array is now a view into atomArray, so
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> r2_cBeta_coords[1] += 1.0  # change the Y coord 1 angstrom
>>> assert r2_cBeta_coords[1] == r2["CB"].coord[1]
>>> # they are always the same (they share the same memory)
>>> r2_cBeta_coords[1] -= 1.0  # restore
```

Note that it is easy to 'break' the view linkage between the Atom coord arrays and the chain atomArray. When modifying Atom coordinates directly, use syntax for an element-by-element copy to avoid this:

```
# use these:
myAtom1.coord[:] = myAtom2.coord
myAtom1.coord[...] = myAtom2.coord
```

```
myAtom1.coord[:] = [1, 2, 3]
for i in range(3):
    myAtom1.coord[i] = myAtom2.coord[i]

# do not use:
myAtom1.coord = myAtom2.coord
myAtom1.coord = [1, 2, 3]
```

Using the `atomArrayIndex` and knowledge of the `AtomKey` class enables us to create Numpy 'selectors', as shown below to extract an array of only the Cα atom coordinates:

```
>>> # create a selector to filter just the C-alpha atoms from the all atom array
>>> atmNameNdx = AtomKey.fields.atm
>>> aaI = myChain.internal_coord.atomArrayIndex
>>> CaSelect = [aaI.get(k) for k in aaI.keys() if k.akl[atmNameNdx] == "CA"]
>>> # now the ordered array of C-alpha atom coordinates is:
>>> CA_coords = myChain.internal_coord.atomArray[CaSelect]
>>> # note this uses Numpy fancy indexing, so CA_coords is a new copy
>>> # (if you modify it, the original atomArray is unaffected)
```

14.6.4.6 Distance Plots

A benefit of the `atomArray` is that generating a distance plot from it is a single line of `Numpy` code:

```
np.linalg.norm(atomArray[:, None, :] - atomArray[None, :, :], axis=-1)
```

Despite its briefness, the idiom cam be difficult to remember and in the form above generates all-atom distances rather than the classic Cα plot as may be desired. The `distance_plot` method wraps the line above and accepts an optional selector like `CaSelect` defined in the previous section. See Figure 14.2.

```
# create a C-alpha distance plot
caDistances = myChain.internal_coord.distance_plot(CaSelect)
# display with e.g. MatPlotLib:
import matplotlib.pyplot as plt

plt.imshow(caDistances, cmap="hot", interpolation="nearest")
plt.show()
```

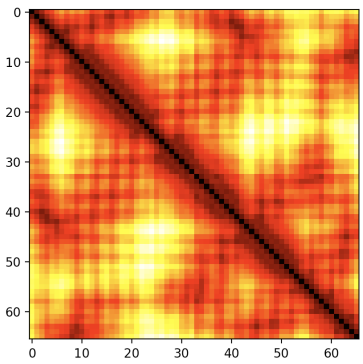


Figure 14.2: Cα distance plot for PDB file 1A8O (HIV capsid C-terminal domain)

14.6.4.7 Building a structure from a distance plot

The all-atom distance plot is another representation of a protein structure, also invariant to translation and rotation but lacking in chirality information (a mirror-image structure will generate the same distance plot). By combining the distance matrix with the signs of each dihedral angle, it is possible to regenerate the internal coordinates.

This work uses equations developed by Blue, the Hedronometer, discussed in https://math.stackexchange.com/a/49340/409 and further in http://daylateanddollarshort.com/mathdocs/Heron-like-Results-for-Tetrahedral.pdf.

To begin, we extract the distances and chirality values from 'myChain':

```
>>> # build structure from distance plot:

>>> ## create the all-atom distance plot
>>> distances = myCic.distance_plot()
>>> ## get the signs of the dihedral angles
>>> chirality = myCic.dihedral_signs()
```

We need a valid data structure matching 'myChain' to correctly rebuild it; using `read_PIC_seq` above would work in the general case, but the 1A8O example used here has some ALTLOC complexity which the sequence alone would not generate. For demonstration the easiest approach is to simply duplicate the 'myChain' structure, but we set all the atom and internal coordinate chain arrays to 0s (only for demonstration) just to be certain there is no data coming through from the original structure:

```
>>> ## get new, empty data structure : copy data structure from myChain
>>> myChain2 = IC_duplicate(myChain)[0]["A"]
>>> cic2 = myChain2.internal_coord

>>> ## clear the new atomArray and di/hedra value arrays, just for proof
>>> cic2.atomArray = np.zeros((cic2.AAsiz, 4), dtype=np.float64)
>>> cic2.dihedraAngle[:] = 0.0
>>> cic2.hedraAngle[:] = 0.0
>>> cic2.hedraL12[:] = 0.0
>>> cic2.hedraL23[:] = 0.0
```

The approach is to regenerate the internal coordinates from the distance plot data, then generate the atom coordinates from the internal coordinates as shown above. To place the final generated structure in the same coordinate space as the starting structure, we copy just the coordinates for the first three N-Cα-C atoms from the chain start of 'myChain' to the 'myChain2' structure (this is only needed to demonstrate equivalence at end):

```
>>> ## copy just the first N-Ca-C coords so structures will superimpose:
>>> cic2.copy_initNCaCs(myChain.internal_coord)
```

The `distance_to_internal_coordinates` routine needs arrays of the six inter-atom distances for each dihedron for the target structure. The convenience routine `distplot_to_dh_arrays` extracts these values from the previously generated distance matrix as needed, and may be replaced by a user method to write these data to the arrays in the `Chain.internal_coords` object.

```
>>> ## copy distances to chain arrays:
>>> cic2.distplot_to_dh_arrays(distances, chirality)
>>> ## compute angles and dihedral angles from distances:
>>> cic2.distance_to_internal_coordinates()
```

The steps below generate the atom coordinates from the newly generated 'myChain2' internal coordinates, then use the Numpy `allclose` routine to confirm that all values match to better than PDB file resolution:

```
>>> ## generate XYZ coordinates from internal coordinates:
>>> myChain2.internal_to_atom_coordinates()
>>> ## confirm result atomArray matches original structure:
>>> np.allclose(cic2.atomArray, myCic.atomArray)
True
```

Note that this procedure does not use the entire distance matrix, but only the six local distances between the four atoms of each dihedral angle.

14.6.4.8 Superimposing residues and their neighborhoods

The `internal_coords` module relies on transforming atom coordinates between different coordinate spaces for both calculation of torsion angles and reconstruction of structures. Each dihedron has a coordinate space transform placing its first atom on the XZ plane, second atom at the origin, and third atom on the +Z axis, as well as a corresponding reverse transform which will return it to the coordinates in the original structure. These transform matrices are available to use as shown below. By judicious choice of a reference dihedron, pairwise and higher order residue intereactions can be investigated and visualized across multiple protein structures, e.g. Figure 14.3.

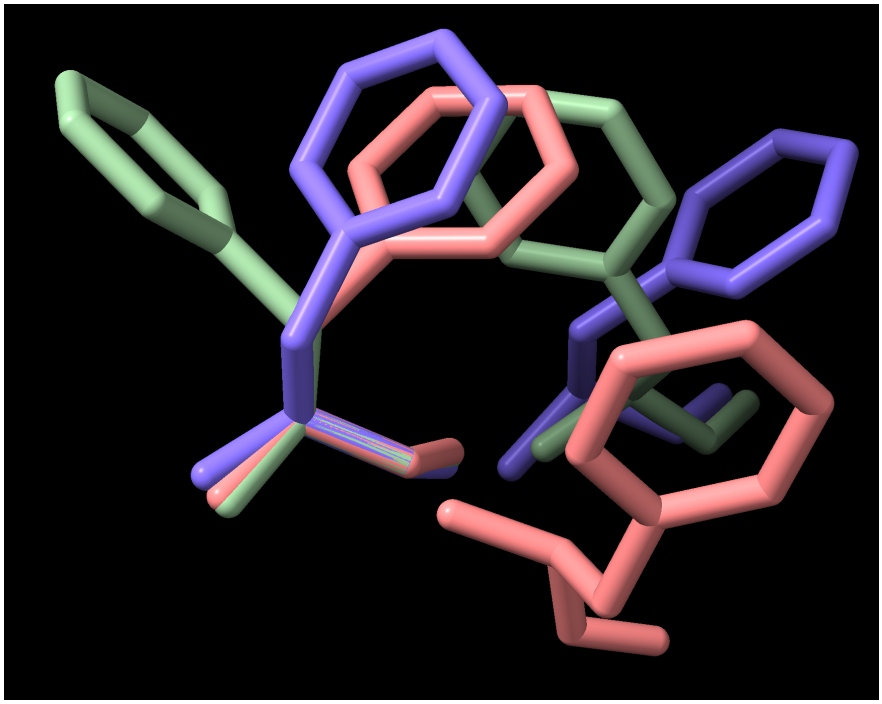


Figure 14.3: Neighboring phenylalanine sidechains in PDB file 3PBL (human dopamine D3 receptor)

This example superimposes each PHE residue in a chain on its N-Cα-Cβ atoms, and presents all PHEs in the chain in the respective coordinate space as a simple demonstration. A more realistic exploration of pairwise sidechain interactions would examine a dataset of structures and filter for interaction classes as discussed in the relevant literature.

```
# superimpose all phe-phe pairs - quick hack just to demonstrate concept
# for analyzing pairwise residue interactions.  Generates PDB ATOM records
```

```
# placing each PHE at origin and showing all other PHEs in environment

## shorthand for key variables:
cic = myChain.internal_coord
resNameNdx = AtomKey.fields.resname
aaNdx = cic.atomArrayIndex

## select just PHE atoms:
pheAtomSelect = [aaNdx.get(k) for k in aaNdx.keys() if k.akl[resNameNdx] == "F"]
aaF = cic.atomArray[pheAtomSelect]  # numpy fancy indexing makes COPY not view

for ric in cic.ordered_aa_ic_list:  # internal_coords version of get_residues()
    if ric.lc == "F":  # if PHE, get transform matrices for chi1 dihedral
        chi1 = ric.pick_angle("chi1")  # N:CA:CB:CG space has C-alpha at origin
        cst = np.transpose(chi1.cst)  # transform TO chi1 space
        # rcst = np.transpose(chi1.rcst)  # transform FROM chi1 space (not needed here)
        cic.atomArray[pheAtomSelect] = aaF.dot(cst)  # transform just the PHEs
        for res in myChain.get_residues():  # print PHEs in new coordinate space
            if res.resname in ["PHE"]:
                print(res.internal_coord.pdb_residue_string())
        cic.atomArray[pheAtomSelect] = aaF  # restore coordinate space from copy
```

14.6.4.9 3D printing protein structures

OpenSCAD (https://openscad.org) is a language for creating solid 3D CAD objects. The algorithm to construct a protein structure from internal coordinates is supplied in OpenSCAD with data describing a structure, such that a model can be generated suitable for 3D printing. While other software can generate STL data as a rendering option for 3D printing (e.g. Chimera, https://www.cgl.ucsf.edu/chimera/), this approach generates spheres and cylinders as output and is therefore more amenable to modifications relevant to 3D printing protein structures. Individual residues and bonds can be selected in the OpenSCAD code for special handling, such as highlighting by size or adding rotatable bonds in specific positions (see https://www.thingiverse.com/thing:3957471 for an example).

```
# write OpenSCAD program of spheres and cylinders to 3d print myChain backbone
## set atom load filter to accept backbone only:
IC_Residue.accept_atoms = IC_Residue.accept_backbone
## set chain break cutoff very high to bridge missing residues with long bonds
IC_Chain.MaxPeptideBond = 4.0
## delete existing data to force re-read of all atoms with attributes set above:
myChain.internal_coord = None
write_SCAD(myChain, "myChain.scad", scale=10.0)
```

14.6.4.10 `internal_coords` control attributes

A few control attributes are available in the `internal_coords` classes to modify or filter data as internal coordinates are calculated. These are listed in Table 14.1:

14.6.5 Determining atom-atom contacts

Use `NeighborSearch` to perform neighbor lookup. The neighbor lookup is done using a KD tree module written in C (see the `KDTree` class in module `Bio.PDB.kdtrees`), making it very fast. It also includes a fast method to find all point pairs within a certain distance of each other.

Class	Attribute	Default	Effect
AtomKey	d2h	False	Convert D atoms to H if True
IC_Chain	MaxPeptideBond	1.4	Max C-N length w/o chain break; make large to link over
IC_Residue	accept_atoms	mainchain, hydrogen atoms	override to remove some or all sidechains, H's, D's
	accept_resnames	CYG, YCM, UNK	3-letter names for HETATMs to process, backbone only u
	gly_Cbeta	False	override to generate Gly Cβ atoms based on database ave

Table 14.1: Control attributes in Bio.PDB.internal_coords.

14.6.6 Superimposing two structures

Use a `Superimposer` object to superimpose two coordinate sets. This object calculates the rotation and translation matrix that rotates two lists of atoms on top of each other in such a way that their RMSD is minimized. Of course, the two lists need to contain the same number of atoms. The `Superimposer` object can also apply the rotation/translation to a list of atoms. The rotation and translation are stored as a tuple in the `rotran` attribute of the `Superimposer` object (note that the rotation is right multiplying!). The RMSD is stored in the `rmsd` attribute.

The algorithm used by `Superimposer` comes from [15, Golub & Van Loan] and makes use of singular value decomposition (this is implemented in the general `Bio.SVDSuperimposer` module).

Example:

```
>>> sup = Superimposer()
# Specify the atom lists
# 'fixed' and 'moving' are lists of Atom objects
# The moving atoms will be put on the fixed atoms
>>> sup.set_atoms(fixed, moving)
# Print rotation/translation/rmsd
>>> print(sup.rotran)
>>> print(sup.rms)
# Apply rotation/translation to the moving atoms
>>> sup.apply(moving)
```

To superimpose two structures based on their active sites, use the active site atoms to calculate the rotation/translation matrices (as above), and apply these to the whole molecule.

14.6.7 Mapping the residues of two related structures onto each other

First, create an alignment file in FASTA format, then use the `StructureAlignment` class. This class can also be used for alignments with more than two structures.

14.6.8 Calculating the Half Sphere Exposure

Half Sphere Exposure (HSE) is a new, 2D measure of solvent exposure [18]. Basically, it counts the number of Cα atoms around a residue in the direction of its side chain, and in the opposite direction (within a radius of 13 Å. Despite its simplicity, it outperforms many other measures of solvent exposure.

HSE comes in two flavors: HSEα and HSEβ. The former only uses the Cα atom positions, while the latter uses the Cα and Cβ atom positions. The HSE measure is calculated by the `HSExposure` class, which can also calculate the contact number. The latter class has methods which return dictionaries that map a `Residue` object to its corresponding HSEα, HSEβ and contact number values.

Example:

```
>>> model = structure[0]
>>> hse = HSExposure()
```

Code	Secondary structure
H	α-helix
B	Isolated β-bridge residue
E	Strand
G	3-10 helix
I	II-helix
T	Turn
S	Bend
-	Other

Table 14.2: DSSP codes in Bio.PDB.

```
# Calculate HSEalpha
>>> exp_ca = hse.calc_hs_exposure(model, option="CA3")
# Calculate HSEbeta
>>> exp_cb = hse.calc_hs_exposure(model, option="CB")
# Calculate classical coordination number
>>> exp_fs = hse.calc_fs_exposure(model)
# Print HSEalpha for a residue
>>> print(exp_ca[some_residue])
```

14.6.9 Determining the secondary structure

For this functionality, you need to install DSSP (and obtain a license for it — free for academic use, see https://swift.cmbi.umcn.nl/gv/dssp/). Then use the `DSSP` class, which maps `Residue` objects to their secondary structure (and accessible surface area). The DSSP codes are listed in Table 14.2. Note that DSSP (the program, and thus by consequence the class) cannot handle multiple models!

The `DSSP` class can also be used to calculate the accessible surface area of a residue. But see also section 14.6.10.

14.6.10 Calculating the residue depth

Residue depth is the average distance of a residue's atoms from the solvent accessible surface. It's a fairly new and very powerful parameterization of solvent accessibility. For this functionality, you need to install Michel Sanner's MSMS program (https://www.scripps.edu/sanner/html/msms_home.html). Then use the `ResidueDepth` class. This class behaves as a dictionary which maps `Residue` objects to corresponding (residue depth, Cα depth) tuples. The Cα depth is the distance of a residue's Cα atom to the solvent accessible surface.

Example:

```
>>> model = structure[0]
>>> rd = ResidueDepth(model, pdb_file)
>>> residue_depth, ca_depth = rd[some_residue]
```

You can also get access to the molecular surface itself (via the `get_surface` function), in the form of a Numeric Python array with the surface points.

14.7 Common problems in PDB files

It is well known that many PDB files contain semantic errors (not the structures themselves, but their representation in PDB files). Bio.PDB tries to handle this in two ways. The PDBParser object can behave in two ways: a restrictive way and a permissive way, which is the default.

Example:

```
# Permissive parser
>>> parser = PDBParser(PERMISSIVE=1)
>>> parser = PDBParser()  # The same (default)
# Strict parser
>>> strict_parser = PDBParser(PERMISSIVE=0)
```

In the permissive state (DEFAULT), PDB files that obviously contain errors are "corrected" (i.e. some residues or atoms are left out). These errors include:

- Multiple residues with the same identifier
- Multiple atoms with the same identifier (taking into account the altloc identifier)

These errors indicate real problems in the PDB file (for details see [16, Hamelryck and Manderick, 2003]). In the restrictive state, PDB files with errors cause an exception to occur. This is useful to find errors in PDB files.

Some errors however are automatically corrected. Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are 'broken'. This is also correctly interpreted.

14.7.1 Examples

The PDBParser/Structure class was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC.

Three exceptions were generated in cases where an unambiguous data structure could not be built. In all three cases, the likely cause is an error in the PDB file that should be corrected. Generating an exception in these cases is much better than running the chance of incorrectly describing the structure in a data structure.

14.7.1.1 Duplicate residues

One structure contains two amino acid residues in one chain with the same sequence identifier (resseq 3) and icode. Upon inspection it was found that this chain contains the residues Thr A3, ..., Gly A202, Leu A3, Glu A204. Clearly, Leu A3 should be Leu A203. A couple of similar situations exist for structure 1FFK (which e.g. contains Gly B64, Met B65, Glu B65, Thr B67, i.e. residue Glu B65 should be Glu B66).

14.7.1.2 Duplicate atoms

Structure 1EJG contains a Ser/Pro point mutation in chain A at position 22. In turn, Ser 22 contains some disordered atoms. As expected, all atoms belonging to Ser 22 have a non-blank altloc specifier (B or C). All atoms of Pro 22 have altloc A, except the N atom which has a blank altloc. This generates an exception, because all atoms belonging to two residues at a point mutation should have non-blank altloc. It turns out that this atom is probably shared by Ser and Pro 22, as Ser 22 misses the N atom. Again, this points to a problem in the file: the N atom should be present in both the Ser and the Pro residue, in both cases associated with a suitable altloc identifier.

14.7.2 Automatic correction

Some errors are quite common and can be easily corrected without much risk of making a wrong interpretation. These cases are listed below.

14.7.2.1 A blank altloc for a disordered atom

Normally each disordered atom should have a non-blank altloc identifier. However, there are many structures that do not follow this convention, and have a blank and a non-blank identifier for two disordered positions of the same atom. This is automatically interpreted in the right way.

14.7.2.2 Broken chains

Sometimes a structure contains a list of residues belonging to chain A, followed by residues belonging to chain B, and again followed by residues belonging to chain A, i.e. the chains are "broken". This is correctly interpreted.

14.7.3 Fatal errors

Sometimes a PDB file cannot be unambiguously interpreted. Rather than guessing and risking a mistake, an exception is generated, and the user is expected to correct the PDB file. These cases are listed below.

14.7.3.1 Duplicate residues

All residues in a chain should have a unique id. This id is generated based on:

- The sequence identifier (resseq).
- The insertion code (icode).
- The hetfield string ("W" for waters and "H_" followed by the residue name for other hetero residues)
- The residue names of the residues in the case of point mutations (to store the Residue objects in a DisorderedResidue object).

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

14.7.3.2 Duplicate atoms

All atoms in a residue should have a unique id. This id is generated based on:

- The atom name (without spaces, or with spaces if a problem arises).
- The altloc specifier.

If this does not lead to a unique id something is quite likely wrong, and an exception is generated.

14.8 Accessing the Protein Data Bank

14.8.1 Downloading structures from the Protein Data Bank

Structures can be downloaded from the PDB (Protein Data Bank) by using the `retrieve_pdb_file` method on a `PDBList` object. The argument for this method is the PDB identifier of the structure.

```
>>> pdbl = PDBList()
>>> pdbl.retrieve_pdb_file("1FAT")
```

The PDBList class can also be used as a command-line tool:

```
python PDBList.py 1fat
```

The downloaded file will be called pdb1fat.ent and stored in the current working directory. Note that the retrieve_pdb_file method also has an optional argument pdir that specifies a specific directory in which to store the downloaded PDB files.

The retrieve_pdb_file method also has some options to specify the compression format used for the download, and the program used for local decompression (default .Z format and gunzip). In addition, the PDB ftp site can be specified upon creation of the PDBList object. By default, the server of the Worldwide Protein Data Bank (ftp://ftp.wwpdb.org/pub/pdb/data/structures/divided/pdb/) is used. See the API documentation for more details. Thanks again to Kristian Rother for donating this module.

14.8.2 Downloading the entire PDB

The following commands will store all PDB files in the /data/pdb directory:

```
python PDBList.py all /data/pdb
```

```
python PDBList.py all /data/pdb -d
```

The API method for this is called download_entire_pdb. Adding the -d option will store all files in the same directory. Otherwise, they are sorted into PDB-style subdirectories according to their PDB ID's. Depending on the traffic, a complete download will take 2-4 days.

14.8.3 Keeping a local copy of the PDB up to date

This can also be done using the PDBList object. One simply creates a PDBList object (specifying the directory where the local copy of the PDB is present) and calls the update_pdb method:

```
>>> pl = PDBList(pdb="/data/pdb")
>>> pl.update_pdb()
```

One can of course make a weekly cronjob out of this to keep the local copy automatically up-to-date. The PDB ftp site can also be specified (see API documentation).

PDBList has some additional methods that can be of use. The get_all_obsolete method can be used to get a list of all obsolete PDB entries. The changed_this_week method can be used to obtain the entries that were added, modified or obsoleted during the current week. For more info on the possibilities of PDBList, see the API documentation.

14.9 General questions

14.9.1 How well tested is Bio.PDB?

Pretty well, actually. Bio.PDB has been extensively tested on nearly 5500 structures from the PDB - all structures seemed to be parsed correctly. More details can be found in the Bio.PDB Bioinformatics article. Bio.PDB has been used/is being used in many research projects as a reliable tool. In fact, I'm using Bio.PDB almost daily for research purposes and continue working on improving it and adding new features.

14.9.2 How fast is it?

The PDBParser performance was tested on about 800 structures (each belonging to a unique SCOP superfamily). This takes about 20 minutes, or on average 1.5 seconds per structure. Parsing the structure of the large ribosomal subunit (1FKK), which contains about 64000 atoms, takes 10 seconds on a 1000 MHz PC. In short: it's more than fast enough for many applications.

14.9.3 Is there support for molecular graphics?

Not directly, mostly since there are quite a few Python based/Python aware solutions already, that can potentially be used with Bio.PDB. My choice is Pymol, BTW (I've used this successfully with Bio.PDB, and there will probably be specific PyMol modules in Bio.PDB soon/some day). Python based/aware molecular graphics solutions include:

- PyMol: https://pymol.org/
- Chimera: https://www.cgl.ucsf.edu/chimera/
- PMV: http://www.scripps.edu/~sanner/python/
- Coot: https://www2.mrc-lmb.cam.ac.uk/personal/pemsley/coot/
- CCP4mg: http://www.ccp4.ac.uk/MG/
- mmLib: http://pymmlib.sourceforge.net/
- VMD: https://www.ks.uiuc.edu/Research/vmd/
- MMTK: http://dirac.cnrs-orleans.fr/MMTK/

14.9.4 Who's using Bio.PDB?

Bio.PDB was used in the construction of DISEMBL, a web server that predicts disordered regions in proteins (http://dis.embl.de/). Bio.PDB has also been used to perform a large scale search for active sites similarities between protein structures in the PDB [17, Hamelryck, 2003], and to develop a new algorithm that identifies linear secondary structure elements [31, Majumdar *et al.*, 2005].

Judging from requests for features and information, Bio.PDB is also used by several LPCs (Large Pharmaceutical Companies :-).

Chapter 15

Bio.PopGen: Population genetics

`Bio.PopGen` is a Biopython module supporting population genetics, available in Biopython 1.44 onwards. The objective for the module is to support widely used data formats, applications and databases.

15.1 GenePop

GenePop (http://genepop.curtin.edu.au/) is a popular population genetics software package supporting Hardy-Weinberg tests, linkage disequilibrium, population differentiation, basic statistics, F_{st} and migration estimates, among others. GenePop does not supply sequence based statistics as it doesn't handle sequence data. The GenePop file format is supported by a wide range of other population genetic software applications, thus making it a relevant format in the population genetics field.

`Bio.PopGen` provides a parser and generator of GenePop file format. Utilities to manipulate the content of a record are also provided. Here is an example on how to read a GenePop file (you can find example GenePop data files in the Test/PopGen directory of Biopython):

```
from Bio.PopGen import GenePop

with open("example.gen") as handle:
    rec = GenePop.read(handle)
```

This will read a file called example.gen and parse it. If you do print rec, the record will be output again, in GenePop format.

The most important information in rec will be the loci names and population information (but there is more – use help(GenePop.Record) to check the API documentation). Loci names can be found on rec.loci_list. Population information can be found on rec.populations. Populations is a list with one element per population. Each element is itself a list of individuals, each individual is a pair composed by individual name and a list of alleles (2 per marker), here is an example for rec.populations:

```
[
    [
        ("Ind1", [(1, 2), (3, 3), (200, 201)]),
        ("Ind2", [(2, None), (3, 3), (None, None)]),
    ],
    [
        ("Other1", [(1, 1), (4, 3), (200, 200)]),
    ],
]
```

So we have two populations, the first with two individuals, the second with only one. The first individual of the first population is called Ind1, allelic information for each of the 3 loci follows. Please note that for any locus, information might be missing (see as an example, Ind2 above).

A few utility functions to manipulate GenePop records are made available, here is an example:

```
from Bio.PopGen import GenePop

# Imagine that you have loaded rec, as per the code snippet above...

rec.remove_population(pos)
# Removes a population from a record, pos is the population position in
# rec.populations, remember that it starts on position 0.
# rec is altered.

rec.remove_locus_by_position(pos)
# Removes a locus by its position, pos is the locus position in
#  rec.loci_list, remember that it starts on position 0.
#  rec is altered.

rec.remove_locus_by_name(name)
# Removes a locus by its name, name is the locus name as in
# rec.loci_list. If the name doesn't exist the function fails
# silently.
# rec is altered.

rec_loci = rec.split_in_loci()
# Splits a record in loci, that is, for each loci, it creates a new
# record, with a single loci and all populations.
# The result is returned in a dictionary, being each key the locus name.
# The value is the GenePop record.
# rec is not altered.

rec_pops = rec.split_in_pops(pop_names)
# Splits a record in populations, that is, for each population, it creates
# a new record, with a single population and all loci.
# The result is returned in a dictionary, being each key
# the population name. As population names are not available in GenePop,
# they are passed in array (pop_names).
# The value of each dictionary entry is the GenePop record.
# rec is not altered.
```

GenePop does not support population names, a limitation which can be cumbersome at times. Functionality to enable population names is currently being planned for Biopython. These extensions won't break compatibility in any way with the standard format. In the medium term, we would also like to support the GenePop web service.

Chapter 16

Phylogenetics with Bio.Phylo

The Bio.Phylo module was introduced in Biopython 1.54. Following the lead of SeqIO and AlignIO, it aims to provide a common way to work with phylogenetic trees independently of the source data format, as well as a consistent API for I/O operations.

Bio.Phylo is described in an open-access journal article [45, Talevich *et al.*, 2012], which you might also find helpful.

16.1 Demo: What's in a Tree?

To get acquainted with the module, let's start with a tree that we've already constructed, and inspect it a few different ways. Then we'll colorize the branches, to use a special phyloXML feature, and finally save it.

Create a simple Newick file named `simple.dnd` using your favorite text editor, or use `simple.dnd` provided with the Biopython source code:

```
(((A,B),(C,D)),(E,F,G));
```

This tree has no branch lengths, only a topology and labeled terminals. (If you have a real tree file available, you can follow this demo using that instead.)

Launch the Python interpreter of your choice:

```
$ ipython -pylab
```

For interactive work, launching the IPython interpreter with the `-pylab` flag enables **matplotlib** integration, so graphics will pop up automatically. We'll use that during this demo.

Now, within Python, read the tree file, giving the file name and the name of the format.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
```

Printing the tree object as a string gives us a look at the entire object hierarchy.

```
>>> print(tree)
Tree(rooted=False, weight=1.0)
    Clade()
        Clade()
            Clade()
                Clade(name='A')
                Clade(name='B')
            Clade()
```

```
            Clade(name='C')
            Clade(name='D')
        Clade()
            Clade(name='E')
            Clade(name='F')
            Clade(name='G')
```

The `Tree` object contains global information about the tree, such as whether it's rooted or unrooted. It has one root clade, and under that, it's nested lists of clades all the way down to the tips.

The function `draw_ascii` creates a simple ASCII-art (plain text) dendrogram. This is a convenient visualization for interactive exploration, in case better graphical tools aren't available.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
>>> Phylo.draw_ascii(tree)
                                                 ____________________ A
                             ____________________|
                            |                    |____________________ B
  __________________________|
 |                          |                     ____________________ C
 |                          |____________________|
_|                                               |____________________ D
 |
 |                           ____________________ E
 |                          |
 |__________________________|____________________ F
                            |
                            |____________________ G
<BLANKLINE>
```

If you have **matplotlib** or **pylab** installed, you can create a graphical tree using the `draw` function.

```
>>> tree.rooted = True
```

```
>>> Phylo.draw(tree)
```

See Figure 16.1.

16.1.1 Coloring branches within a tree

The function `draw` supports the display of different colors and branch widths in a tree. As of Biopython 1.59, the `color` and `width` attributes are available on the basic Clade object and there's nothing extra required to use them. Both attributes refer to the branch leading the given clade, and apply recursively, so all descendent branches will also inherit the assigned width and color values during display.

In earlier versions of Biopython, these were special features of PhyloXML trees, and using the attributes required first converting the tree to a subclass of the basic tree object called Phylogeny, from the Bio.Phylo.PhyloXML module.

In Biopython 1.55 and later, this is a convenient tree method:

```
>>> tree = tree.as_phyloxml()
```

In Biopython 1.54, you can accomplish the same thing with one extra import:

```
>>> from Bio.Phylo.PhyloXML import Phylogeny
>>> tree = Phylogeny.from_tree(tree)
```

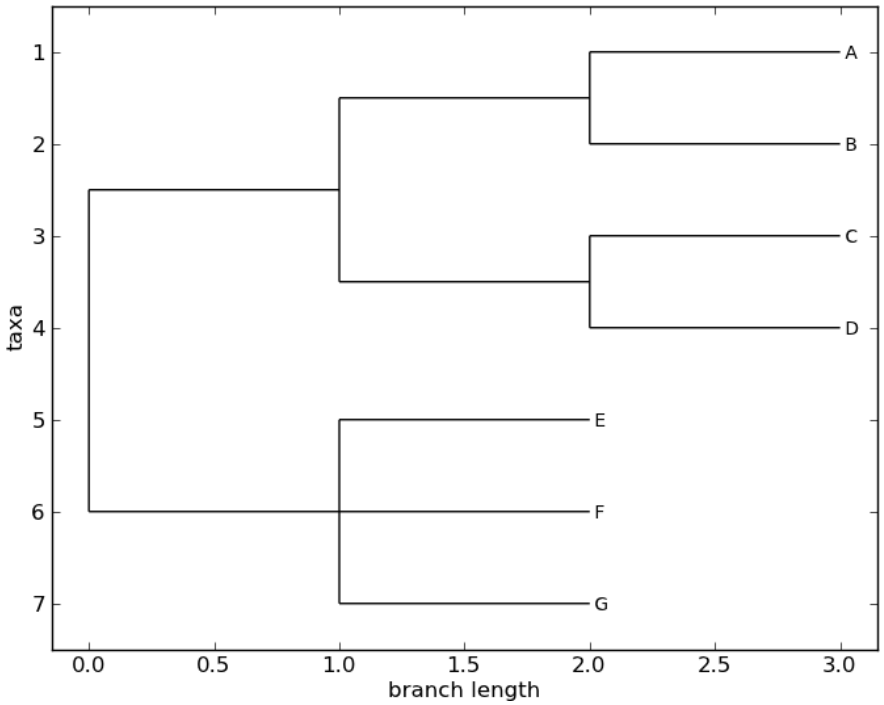


Figure 16.1: A rooted tree drawn with `Phylo.draw`.

Note that the file formats Newick and Nexus don't support branch colors or widths, so if you use these attributes in Bio.Phylo, you will only be able to save the values in PhyloXML format. (You can still save a tree as Newick or Nexus, but the color and width values will be skipped in the output file.)

Now we can begin assigning colors. First, we'll color the root clade gray. We can do that by assigning the 24-bit color value as an RGB triple, an HTML-style hex string, or the name of one of the predefined colors.

```
>>> tree.root.color = (128, 128, 128)
```

Or:

```
>>> tree.root.color = "#808080"
```

Or:

```
>>> tree.root.color = "gray"
```

Colors for a clade are treated as cascading down through the entire clade, so when we colorize the root here, it turns the whole tree gray. We can override that by assigning a different color lower down on the tree.

Let's target the most recent common ancestor (MRCA) of the nodes named "E" and "F". The `common_ancestor` method returns a reference to that clade in the original tree, so when we color that clade "salmon", the color will show up in the original tree.

```
>>> mrca = tree.common_ancestor({"name": "E"}, {"name": "F"})
>>> mrca.color = "salmon"
```

If we happened to know exactly where a certain clade is in the tree, in terms of nested list entries, we can jump directly to that position in the tree by indexing it. Here, the index `[0,1]` refers to the second child of the first child of the root.

```
>>> tree.clade[0, 1].color = "blue"
```

Finally, show our work:

```
>>> Phylo.draw(tree)
```

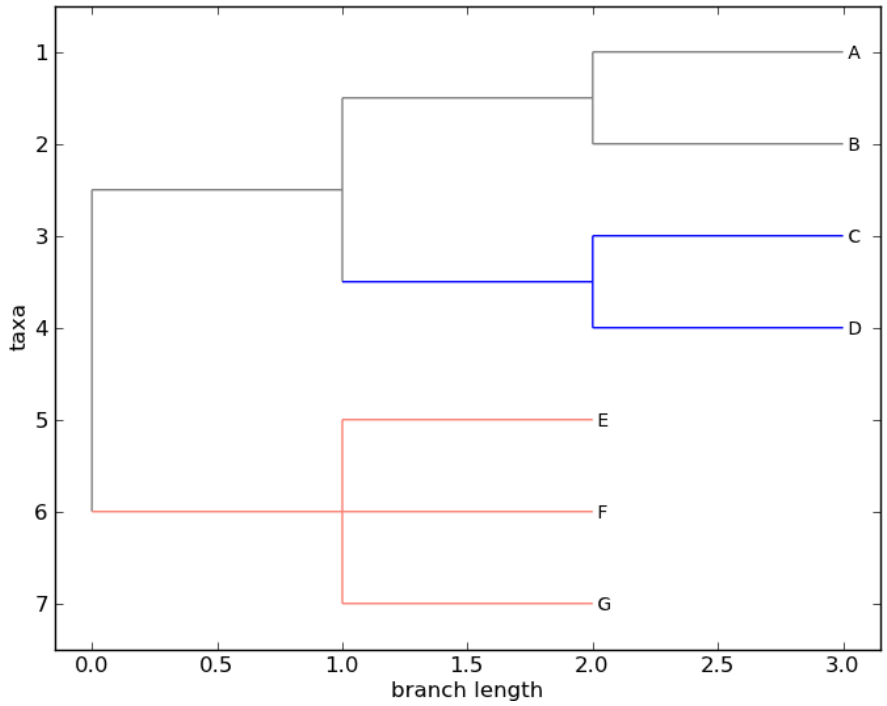


Figure 16.2: A colorized tree drawn with `Phylo.draw`.

See Figure 16.2.

Note that a clade's color includes the branch leading to that clade, as well as its descendents. The common ancestor of E and F turns out to be just under the root, and with this coloring we can see exactly where the root of the tree is.

My, we've accomplished a lot! Let's take a break here and save our work. Call the `write` function with a file name or handle — here we use standard output, to see what would be written — and the format `phyloxml`. PhyloXML saves the colors we assigned, so you can open this phyloXML file in another tree viewer like Archaeopteryx, and the colors will show up there, too.

```
>>> import sys
>>> n = Phylo.write(tree, sys.stdout, "phyloxml")  # doctest:+ELLIPSIS
<phyloxml ...>
  <phylogeny rooted="true">
    <clade>
      <color>
        <red>128</red>
        <green>128</green>
        <blue>128</blue>
      </color>
      <clade>
        <clade>
```

```
          <clade>
            <name>A</name>
          </clade>
          <clade>
            <name>B</name>
          </clade>
        </clade>
        <clade>
          <color>
            <red>0</red>
            <green>0</green>
            <blue>255</blue>
          </color>
          <clade>
            <name>C</name>
          </clade>
          ...
    </clade>
  </phylogeny>
</phyloxml>
>>> n
1
```

The rest of this chapter covers the core functionality of Bio.Phylo in greater detail. For more examples of using Bio.Phylo, see the cookbook page on Biopython.org:
http://biopython.org/wiki/Phylo_cookbook

16.2 I/O functions

Like SeqIO and AlignIO, Phylo handles file input and output through four functions: `parse`, `read`, `write` and `convert`, all of which support the tree file formats Newick, NEXUS, phyloXML and NeXML, as well as the Comparative Data Analysis Ontology (CDAO).

The `read` function parses a single tree in the given file and returns it. Careful; it will raise an error if the file contains more than one tree, or no trees.

```
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Nexus/int_node_labels.nwk", "newick")
>>> print(tree)  # doctest:+ELLIPSIS
Tree(rooted=False, weight=1.0)
    Clade(branch_length=75.0, name='gymnosperm')
        Clade(branch_length=25.0, name='Coniferales')
            Clade(branch_length=25.0)
                Clade(branch_length=10.0, name='Tax+nonSci')
                    Clade(branch_length=90.0, name='Taxaceae')
                        Clade(branch_length=125.0, name='Cephalotaxus')
                        ...
```

(Example files are available in the `Tests/Nexus/` and `Tests/PhyloXML/` directories of the Biopython distribution.)

To handle multiple (or an unknown number of) trees, use the `parse` function iterates through each of the trees in the given file:

```
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> for tree in trees:
...     print(tree)  # doctest:+ELLIPSIS
...
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...', name='example fro
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')
            ...
```

Write a tree or iterable of trees back to file with the `write` function:

```
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> tree1 = next(trees)
>>> Phylo.write(tree1, "tree1.nwk", "newick")
1
>>> Phylo.write(trees, "other_trees.xml", "phyloxml")  # write the remaining trees
12
```

Convert files between any of the supported formats with the `convert` function:

```
>>> Phylo.convert("tree1.nwk", "newick", "tree1.xml", "nexml")
1
>>> Phylo.convert("other_trees.xml", "phyloxml", "other_trees.nex", "nexus")
12
```

To use strings as input or output instead of actual files, use `StringIO` as you would with SeqIO and AlignIO:

```
>>> from Bio import Phylo
>>> from io import StringIO
>>> handle = StringIO("(((A,B),(C,D)),(E,F,G));")
>>> tree = Phylo.read(handle, "newick")
```

16.3 View and export trees

The simplest way to get an overview of a `Tree` object is to `print` it:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> print(tree)
Phylogeny(description='phyloXML allows to use either a "branch_length" attribute...', name='example fro
    Clade()
        Clade(branch_length=0.06)
            Clade(branch_length=0.102, name='A')
            Clade(branch_length=0.23, name='B')
        Clade(branch_length=0.4, name='C')
```

This is essentially an outline of the object hierarchy Biopython uses to represent a tree. But more likely, you'd want to see a drawing of the tree. There are three functions to do this.

As we saw in the demo, `draw_ascii` prints an ascii-art drawing of the tree (a rooted phylogram) to standard output, or an open file handle if given. Not all of the available information about the tree is shown, but it provides a way to quickly view the tree without relying on any external dependencies.

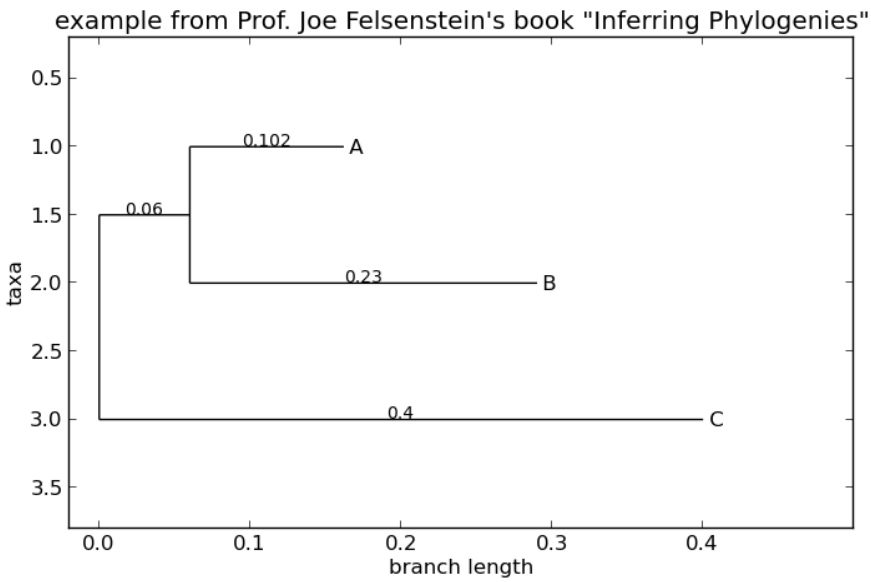


Figure 16.3: A simple rooted tree plotted with the `draw` function.

```
>>> tree = Phylo.read("PhyloXML/example.xml", "phyloxml")
>>> Phylo.draw_ascii(tree)
              ________________ A
  ___________|
_|           |_____________________________________ B
 |
 |_____________________________________________________________________________ C
<BLANKLINE>
```

The `draw` function draws a more attractive image using the matplotlib library. See the API documentation for details on the arguments it accepts to customize the output.

```
>>> Phylo.draw(tree, branch_labels=lambda c: c.branch_length)
```

See Figure 16.3 for example.

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for descriptions and examples of the more advanced functionality in `draw_ascii`, `draw_graphviz` and `to_networkx`.

16.4 Using Tree and Clade objects

The `Tree` objects produced by `parse` and `read` are containers for recursive sub-trees, attached to the `Tree` object at the `root` attribute (whether or not the phylogenetic tree is actually considered rooted). A `Tree` has globally applied information for the phylogeny, such as rootedness, and a reference to a single `Clade`; a `Clade` has node- and clade-specific information, such as branch length, and a list of its own descendent `Clade` instances, attached at the `clades` attribute.

So there is a distinction between `tree` and `tree.root`. In practice, though, you rarely need to worry about it. To smooth over the difference, both `Tree` and `Clade` inherit from `TreeMixin`, which contains the implementations for methods that would be commonly used to search, inspect or modify a tree or any of its clades. This means that almost all of the methods supported by `tree` are also available on `tree.root` and any clade below it. (`Clade` also has a `root` property, which returns the clade object itself.)

16.4.1 Search and traversal methods

For convenience, we provide a couple of simplified methods that return all external or internal nodes directly as a list:

`get_terminals` makes a list of all of this tree's terminal (leaf) nodes.

`get_nonterminals` makes a list of all of this tree's nonterminal (internal) nodes.

These both wrap a method with full control over tree traversal, `find_clades`. Two more traversal methods, `find_elements` and `find_any`, rely on the same core functionality and accept the same arguments, which we'll call a "target specification" for lack of a better description. These specify which objects in the tree will be matched and returned during iteration. The first argument can be any of the following types:

- A **TreeElement instance**, which tree elements will match by identity — so searching with a Clade instance as the target will find that clade in the tree;
- A **string**, which matches tree elements' string representation — in particular, a clade's `name` *(added in Biopython 1.56)*;
- A **class** or **type**, where every tree element of the same type (or sub-type) will be matched;
- A **dictionary** where keys are tree element attributes and values are matched to the corresponding attribute of each tree element. This one gets even more elaborate:
  - If an `int` is given, it matches numerically equal attributes, e.g. 1 will match 1 or 1.0
  - If a boolean is given (True or False), the corresponding attribute value is evaluated as a boolean and checked for the same
  - `None` matches `None`
  - If a string is given, the value is treated as a regular expression (which must match the whole string in the corresponding element attribute, not just a prefix). A given string without special regex characters will match string attributes exactly, so if you don't use regexes, don't worry about it. For example, in a tree with clade names Foo1, Foo2 and Foo3, `tree.find_clades({"name": "Foo1"})` matches Foo1, `{"name": "Foo.*"}` matches all three clades, and `{"name": "Foo"}` doesn't match anything.

  Since floating-point arithmetic can produce some strange behavior, we don't support matching `float`s directly. Instead, use the boolean `True` to match every element with a nonzero value in the specified attribute, then filter on that attribute manually with an inequality (or exact number, if you like living dangerously).

  If the dictionary contains multiple entries, a matching element must match each of the given attribute values — think "and", not "or".
- A **function** taking a single argument (it will be applied to each element in the tree), returning True or False. For convenience, LookupError, AttributeError and ValueError are silenced, so this provides another safe way to search for floating-point values in the tree, or some more complex characteristic.

After the target, there are two optional keyword arguments:

terminal — A boolean value to select for or against terminal clades (a.k.a. leaf nodes): True searches for only terminal clades, False for non-terminal (internal) clades, and the default, None, searches both terminal and non-terminal clades, as well as any tree elements lacking the `is_terminal` method.

order — Tree traversal order: `"preorder"` (default) is depth-first search, `"postorder"` is DFS with child nodes preceding parents, and `"level"` is breadth-first search.

Finally, the methods accept arbitrary keyword arguments which are treated the same way as a dictionary target specification: keys indicate the name of the element attribute to search for, and the argument value (string, integer, None or boolean) is compared to the value of each attribute found. If no keyword arguments are given, then any TreeElement types are matched. The code for this is generally shorter than passing a dictionary as the target specification: `tree.find_clades({"name": "Foo1"})` can be shortened to `tree.find_clades(name="Foo1")`.

(In Biopython 1.56 or later, this can be even shorter: `tree.find_clades("Foo1")`)

Now that we've mastered target specifications, here are the methods used to traverse a tree:

`find_clades` Find each clade containing a matching element. That is, find each element as with `find_elements`, but return the corresponding clade object. (This is usually what you want.)

The result is an iterable through all matching objects, searching depth-first by default. This is not necessarily the same order as the elements appear in the Newick, Nexus or XML source file!

`find_elements` Find all tree elements matching the given attributes, and return the matching elements themselves. Simple Newick trees don't have complex sub-elements, so this behaves the same as `find_clades` on them. PhyloXML trees often do have complex objects attached to clades, so this method is useful for extracting those.

`find_any` Return the first element found by `find_elements()`, or None. This is also useful for checking whether any matching element exists in the tree, and can be used in a conditional.

Two more methods help navigating between nodes in the tree:

`get_path` List the clades directly between the tree root (or current clade) and the given target. Returns a list of all clade objects along this path, ending with the given target, but excluding the root clade.

`trace` List of all clade object between two targets in this tree. Excluding start, including finish.

16.4.2 Information methods

These methods provide information about the whole tree (or any clade).

`common_ancestor` Find the most recent common ancestor of all the given targets. (This will be a Clade object). If no target is given, returns the root of the current clade (the one this method is called from); if 1 target is given, this returns the target itself. However, if any of the specified targets are not found in the current tree (or clade), an exception is raised.

`count_terminals` Counts the number of terminal (leaf) nodes within the tree.

`depths` Create a mapping of tree clades to depths. The result is a dictionary where the keys are all of the Clade instances in the tree, and the values are the distance from the root to each clade (including terminals). By default the distance is the cumulative branch length leading to the clade, but with the `unit_branch_lengths=True` option, only the number of branches (levels in the tree) is counted.

`distance` Calculate the sum of the branch lengths between two targets. If only one target is specified, the other is the root of this tree.

`total_branch_length` Calculate the sum of all the branch lengths in this tree. This is usually just called the "length" of the tree in phylogenetics, but we use a more explicit name to avoid confusion with Python terminology.

The rest of these methods are boolean checks:

`is_bifurcating` True if the tree is strictly bifurcating; i.e. all nodes have either 2 or 0 children (internal or external, respectively). The root may have 3 descendents and still be considered part of a bifurcating tree.

`is_monophyletic` Test if all of the given targets comprise a complete subclade — i.e., there exists a clade such that its terminals are the same set as the given targets. The targets should be terminals of the tree. For convenience, this method returns the common ancestor (MCRA) of the targets if they are monophyletic (instead of the value `True`), and `False` otherwise.

`is_parent_of` True if target is a descendent of this tree — not required to be a direct descendent. To check direct descendents of a clade, simply use list membership testing: `if subclade in clade: ...`

`is_preterminal` True if all direct descendents are terminal; False if any direct descendent is not terminal.

16.4.3 Modification methods

These methods modify the tree in-place. If you want to keep the original tree intact, make a complete copy of the tree first, using Python's `copy` module:

```
tree = Phylo.read("example.xml", "phyloxml")
import copy

newtree = copy.deepcopy(tree)
```

`collapse` Deletes the target from the tree, relinking its children to its parent.

`collapse_all` Collapse all the descendents of this tree, leaving only terminals. Branch lengths are preserved, i.e. the distance to each terminal stays the same. With a target specification (see above), collapses only the internal nodes matching the specification.

`ladderize` Sort clades in-place according to the number of terminal nodes. Deepest clades are placed last by default. Use `reverse=True` to sort clades deepest-to-shallowest.

`prune` Prunes a terminal clade from the tree. If taxon is from a bifurcation, the connecting node will be collapsed and its branch length added to remaining terminal node. This might no longer be a meaningful value.

`root_with_outgroup` Reroot this tree with the outgroup clade containing the given targets, i.e. the common ancestor of the outgroup. This method is only available on Tree objects, not Clades.

If the outgroup is identical to self.root, no change occurs. If the outgroup clade is terminal (e.g. a single terminal node is given as the outgroup), a new bifurcating root clade is created with a 0-length branch to the given outgroup. Otherwise, the internal node at the base of the outgroup becomes a trifurcating root for the whole tree. If the original root was bifurcating, it is dropped from the tree.

In all cases, the total branch length of the tree stays the same.

`root_at_midpoint` Reroot this tree at the calculated midpoint between the two most distant tips of the tree. (This uses `root_with_outgroup` under the hood.)

`split` Generate n (default 2) new descendants. In a species tree, this is a speciation event. New clades have the given `branch_length` and the same name as this clade's root plus an integer suffix (counting from 0) — for example, splitting a clade named "A" produces the sub-clades "A0" and "A1".

See the Phylo page on the Biopython wiki (http://biopython.org/wiki/Phylo) for more examples of using the available methods.

16.4.4 Features of PhyloXML trees

The phyloXML file format includes fields for annotating trees with additional data types and visual cues.

See the PhyloXML page on the Biopython wiki (http://biopython.org/wiki/PhyloXML) for descriptions and examples of using the additional annotation features provided by PhyloXML.

16.5 Running external applications

While Bio.Phylo doesn't infer trees from alignments itself, there are third-party programs available that do. These can be accessed from within python by using the `subprocess` module.

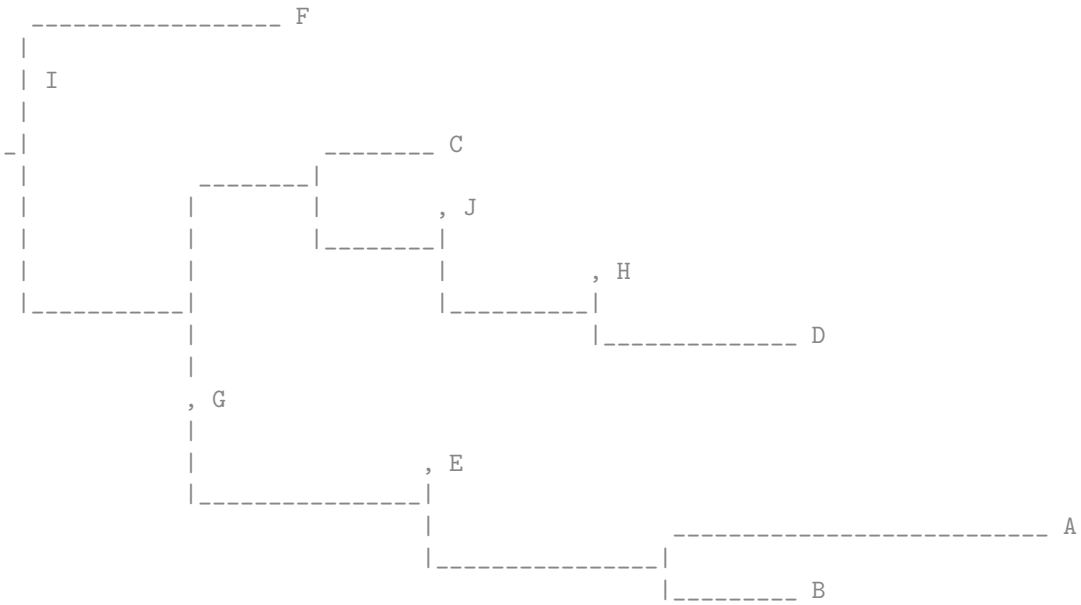
Below is an example on how to use a python script to interact with PhyML (http://www.atgc-montpellier.fr/phyml/). The program accepts an input alignment in `phylip-relaxed` format (that's Phylip format, but without the 10-character limit on taxon names) and a variety of options.

```
>>> import subprocess
>>> cmd = "phyml -i Tests/Phylip/random.phy"
>>> results = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE, text=True)
```

The 'stdout = subprocess.PIPE' argument makes the output of the program accessible through 'results.stdout' for debugging purposes, (the same can be done for 'stderr'), and 'text=True' makes the returned information be a python string, instead of a 'bytes' object.

This generates a tree file and a stats file with the names [*input filename*]`_phyml_tree.txt` and [*input filename*]`_phyml_stats.txt`. The tree file is in Newick format:

```
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Phylip/random.phy_phyml_tree.txt", "newick")
>>> Phylo.draw_ascii(tree)
```




```
<BLANKLINE>
```

The `subprocess` module can also be used for interacting with any other programs that provide a command line interface such as RAxML (https://sco.h-its.org/exelixis/software.html), FastTree (http://www.microbesonline.org/fasttree/), `dnaml` and `protml`.

16.6 PAML integration

Biopython 1.58 brought support for PAML (http://abacus.gene.ucl.ac.uk/software/paml.html), a suite of programs for phylogenetic analysis by maximum likelihood. Currently the programs codeml, baseml and yn00 are implemented. Due to PAML's usage of control files rather than command line arguments to control runtime options, usage of this wrapper strays from the format of other application wrappers in Biopython.

A typical workflow would be to initialize a PAML object, specifying an alignment file, a tree file, an output file and a working directory. Next, runtime options are set via the `set_options()` method or by reading an existing control file. Finally, the program is run via the `run()` method and the output file is automatically parsed to a results dictionary.

Here is an example of typical usage of codeml:

```
>>> from Bio.Phylo.PAML import codeml
>>> cml = codeml.Codeml()
>>> cml.alignment = "Tests/PAML/Alignments/alignment.phylip"
>>> cml.tree = "Tests/PAML/Trees/species.tree"
>>> cml.out_file = "results.out"
>>> cml.working_dir = "./scratch"
>>> cml.set_options(
...     seqtype=1,
...     verbose=0,
...     noisy=0,
...     RateAncestor=0,
...     model=0,
...     NSsites=[0, 1, 2],
...     CodonFreq=2,
...     cleandata=1,
...     fix_alpha=1,
...     kappa=4.54006,
... )
>>> results = cml.run()
>>> ns_sites = results.get("NSsites")
>>> m0 = ns_sites.get(0)
>>> m0_params = m0.get("parameters")
>>> print(m0_params.get("omega"))
```

Existing output files may be parsed as well using a module's `read()` function:

```
>>> results = codeml.read("Tests/PAML/Results/codeml/codeml_NSsites_all.out")
>>> print(results.get("lnL max"))
```

Detailed documentation for this new module currently lives on the Biopython wiki: http://biopython.org/wiki/PAML

16.7 Future plans

Bio.Phylo is under active development. Here are some features we might add in future releases:

New methods Generally useful functions for operating on Tree or Clade objects appear on the Biopython wiki first, so that casual users can test them and decide if they're useful before we add them to Bio.Phylo:

http://biopython.org/wiki/Phylo_cookbook

Bio.Nexus port Much of this module was written during Google Summer of Code 2009, under the auspices of NESCent, as a project to implement Python support for the phyloXML data format (see 16.4.4). Support for Newick and Nexus formats was added by porting part of the existing Bio.Nexus module to the new classes used by Bio.Phylo.

Currently, Bio.Nexus contains some useful features that have not yet been ported to Bio.Phylo classes — notably, calculating a consensus tree. If you find some functionality lacking in Bio.Phylo, try poking through Bio.Nexus to see if it's there instead.

We're open to any suggestions for improving the functionality and usability of this module; just let us know on the mailing list or our bug database.

Finally, if you need additional functionality not yet included in the Phylo module, check if it's available in another of the high-quality Python libraries for phylogenetics such as DendroPy (https://dendropy.org/) or PyCogent (http://pycogent.org/). Since these libraries also support standard file formats for phylogenetic trees, you can easily transfer data between libraries by writing to a temporary file or StringIO object.

Chapter 17

Sequence motif analysis using Bio.motifs

This chapter gives an overview of the functionality of the `Bio.motifs` package included in Biopython. It is intended for people who are involved in the analysis of sequence motifs, so I'll assume that you are familiar with basic notions of motif analysis. In case something is unclear, please look at Section 17.10 for some relevant links.

Most of this chapter describes the new `Bio.motifs` package included in Biopython 1.61 onwards, which is replacing the older `Bio.Motif` package introduced with Biopython 1.50, which was in turn based on two older former Biopython modules, `Bio.AlignAce` and `Bio.MEME`. It provides most of their functionality with a unified motif object implementation.

Speaking of other libraries, if you are reading this you might be interested in TAMO, another python library designed to deal with sequence motifs. It supports more *de-novo* motif finders, but it is not a part of Biopython and has some restrictions on commercial use.

17.1 Motif objects

Since we are interested in motif analysis, we need to take a look at `Motif` objects in the first place. For that we need to import the Bio.motifs library:

```
>>> from Bio import motifs
```

and we can start creating our first motif objects. We can either create a `Motif` object from a list of instances of the motif, or we can obtain a `Motif` object by parsing a file from a motif database or motif finding software.

17.1.1 Creating a motif from instances

Suppose we have these instances of a DNA motif:

```
>>> from Bio.Seq import Seq
>>> instances = [
...     Seq("TACAA"),
...     Seq("TACGC"),
...     Seq("TACAC"),
...     Seq("TACCC"),
...     Seq("AACCC"),
...     Seq("AATGC"),
```

```
...     Seq("AATGC"),
... ]
```

then we can create a Motif object as follows:

```
>>> m = motifs.create(instances)
```

The instances from which this motif was created is stored in the `.alignment` property:

```
>>> print(m.alignment.sequences)
[Seq('TACAA'), Seq('TACGC'), Seq('TACAC'), Seq('TACCC'), Seq('AACCC'), Seq('AATGC'), Seq('AATGC')]
```

Printing the Motif object shows the instances from which it was constructed:

```
>>> print(m)
TACAA
TACGC
TACAC
TACCC
AACCC
AATGC
AATGC
```

The length of the motif is defined as the sequence length, which should be the same for all instances:

```
>>> len(m)
5
```

The Motif object has an attribute `.counts` containing the counts of each nucleotide at each position. Printing this counts matrix shows it in an easily readable format:

```
>>> print(m.counts)
        0      1      2      3      4
A:   3.00   7.00   0.00   2.00   1.00
C:   0.00   0.00   5.00   2.00   6.00
G:   0.00   0.00   0.00   3.00   0.00
T:   4.00   0.00   2.00   0.00   0.00
<BLANKLINE>
```

You can access these counts as a dictionary:

```
>>> m.counts["A"]
[3.0, 7.0, 0.0, 2.0, 1.0]
```

but you can also think of it as a 2D array with the nucleotide as the first dimension and the position as the second dimension:

```
>>> m.counts["T", 0]
4.0
>>> m.counts["T", 2]
2.0
>>> m.counts["T", 3]
0.0
```

You can also directly access columns of the counts matrix

```
>>> m.counts[:, 3]
{'A': 2.0, 'C': 2.0, 'T': 0.0, 'G': 3.0}
```

Instead of the nucleotide itself, you can also use the index of the nucleotide in the alphabet of the motif:

```
>>> m.alphabet
'ACGT'
>>> m.counts["A", :]
(3.0, 7.0, 0.0, 2.0, 1.0)
>>> m.counts[0, :]
(3.0, 7.0, 0.0, 2.0, 1.0)
```

17.1.2 Obtaining a consensus sequence

The consensus sequence of a motif is defined as the sequence of letters along the positions of the motif for which the largest value in the corresponding columns of the `.counts` matrix is obtained:

```
>>> m.consensus
Seq('TACGC')
```

Conversely, the anticonsensus sequence corresponds to the smallest values in the columns of the `.counts` matrix:

```
>>> m.anticonsensus
Seq('CCATG')
```

Note that there is some ambiguity in the definition of the consensus and anticonsensus sequence if in some columns multiple nucleotides have the maximum or minimum count.

For DNA sequences, you can also ask for a degenerate consensus sequence, in which ambiguous nucleotides are used for positions where there are multiple nucleotides with high counts:

```
>>> m.degenerate_consensus
Seq('WACVC')
```

Here, W and R follow the IUPAC nucleotide ambiguity codes: W is either A or T, and V is A, C, or G [7]. The degenerate consensus sequence is constructed following the rules specified by Cavener [3].

The `motif.counts.calculate_consensus` method lets you specify in detail how the consensus sequence should be calculated. This method largely follows the conventions of the EMBOSS program `cons`, and takes the following arguments:

substitution_matrix The scoring matrix used when comparing sequences. By default, it is `None`, in which case we simply count the frequency of each letter. Instead of the default value, you can use the substitution matrices available in `Bio.Align.substitution\_matrices`. Common choices are BLOSUM62 (also known as EBLOSUM62) for protein, and NUC.4.4 (also known as EDNAFULL) for nucleotides. NOTE: Currently, this method has not yet been implemented for values other than the default value `None`.

plurality Threshold value for the number of positive matches, divided by the total count in a column, required to reach consensus. If `substitution_matrix` is `None`, then this argument must also be `None`, and is ignored; a `ValueError` is raised otherwise. If `substitution_matrix` is not `None`, then the default value of the plurality is 0.5.

identity Number of identities, divided by the total count in a column, required to define a consensus value. If the number of identities is less than identity multiplied by the total count in a column, then the undefined character (`N` for nucleotides and `X` for amino acid sequences) is used in the consensus sequence. If `identity` is 1.0, then only columns of identical letters contribute to the consensus. Default value is zero.

setcase threshold for the positive matches, divided by the total count in a column, above which the consensus is is upper-case and below which the consensus is in lower-case. By default, this is equal to 0.5.

This is an example:

```
>>> m.counts.calculate_consensus(identity=0.5, setcase=0.7)
'tACNC'
```

17.1.3 Reverse-complementing a motif

We can get the reverse complement of a motif by calling the `reverse_complement` method on it:

```
>>> r = m.reverse_complement()
>>> r.consensus
Seq('GCGTA')
>>> r.degenerate_consensus
Seq('GBGTW')
>>> print(r)
TTGTA
GCGTA
GTGTA
GGGTA
GGGTT
GCATT
GCATT
```

The reverse complement is only defined for DNA motifs.

17.1.4 Slicing a motif

You can slice the motif to obtain a new `Motif` object for the selected positions:

```
>>> m_sub = m[2:-1]
>>> print(m_sub)
CA
CG
CA
CC
CC
TG
TG
>>> m_sub.consensus
Seq('CG')
>>> m_sub.degenerate_consensus
Seq('CV')
```

17.1.5 Relative entropy

The relative entropy (or Kullback-Leibler distance) H_j of column j of the motif is defined as [39, 11]

$$H_j = \sum_{i=1}^{M} p_{ij} \log\left(\frac{p_{ij}}{b_i}\right)$$

where:

- M – The number of letters in the alphabet (given by `len(m.alphabet)`);
- p_{ij} – The observed frequency of letter i, normalized, in the j-th column (see below);
- b_i – The background probability of letter i (given by `m.background[i]`).

The observed frequency p_{ij} is computed as follows:

$$p_{ij} = \frac{c_{ij} + k_i}{C_j + k}$$

where:

- c_{ij} – the number of times letter i appears in column j of the alignment (given by `m.counts[i, j]`);
- C_j – The total number of letters in column j: $C_j = \sum_{i=1}^{M} c_{ij}$ (given by `sum(m.counts[:, j])`).
- k_i – the pseudocount of letter i (given by `m.pseudocounts[i]`).
- k – the total pseudocount: $k = \sum_{i=1}^{M} k_i$ (given by `sum(m.pseudocounts.values())`).

With these definitions, both p_{ij} and b_i are normalized to 1:

$$\sum_{i=1}^{M} p_{ij} = 1$$

$$\sum_{i=1}^{M} b_i = 1$$

The relative entropy is the same as the information content if the background distribution is uniform.

The relative entropy for each column of motif `m` can be obtained using the `relative_entropy` property:

```
>>> m.relative_entropy
array([1.01477186, 2.        , 1.13687943, 0.44334329, 1.40832722])
```

These values are calculated using the base-2 logarithm, and are therefore in units of bits. The second column (which consists of `A` nucleotides only) has the highest relative entropy; the fourth column (which consists of `A`, `C`, or `G` nucleotides) has the lowest relative entropy). The relative entropy of the motif can be calculated by summing over the columns:

```
>>> sum(m.relative_entropy)  # doctest:+ELLIPSIS
6.003321...
```

17.1.6 Creating a sequence logo

If we have internet access, we can create a weblogo:

```
>>> m.weblogo("mymotif.png")
```

We should get our logo saved as a PNG in the specified file.

17.2 Reading motifs

Creating motifs from instances by hand is a bit boring, so it's useful to have some I/O functions for reading and writing motifs. There are not any really well established standards for storing motifs, but there are a couple of formats that are more used than others.

17.2.1 JASPAR

One of the most popular motif databases is JASPAR. In addition to the motif sequence information, the JASPAR database stores a lot of meta-information for each motif. The module `Bio.motifs` contains a specialized class `jaspar.Motif` in which this meta-information is represented as attributes:

- `matrix_id` - the unique JASPAR motif ID, e.g. 'MA0004.1'
- `name` - the name of the TF, e.g. 'Arnt'
- `collection` - the JASPAR collection to which the motif belongs, e.g. 'CORE'
- `tf_class` - the structural class of this TF, e.g. 'Zipper-Type'
- `tf_family` - the family to which this TF belongs, e.g. 'Helix-Loop-Helix'
- `species` - the species to which this TF belongs, may have multiple values, these are specified as taxonomy IDs, e.g. 10090
- `tax_group` - the taxonomic supergroup to which this motif belongs, e.g. 'vertebrates'
- `acc` - the accession number of the TF protein, e.g. 'P53762'
- `data_type` - the type of data used to construct this motif, e.g. 'SELEX'
- `medline` - the Pubmed ID of literature supporting this motif, may be multiple values, e.g. 7592839
- `pazar_id` - external reference to the TF in the PAZAR database, e.g. 'TF0000003'
- `comment` - free form text containing notes about the construction of the motif

The `jaspar.Motif` class inherits from the generic `Motif` class and therefore provides all the facilities of any of the motif formats — reading motifs, writing motifs, scanning sequences for motif instances etc.

JASPAR stores motifs in several different ways including three different flat file formats and as an SQL database. All of these formats facilitate the construction of a counts matrix. However, the amount of meta information described above that is available varies with the format.

The JASPAR `sites` format

The first of the three flat file formats contains a list of instances. As an example, these are the beginning and ending lines of the JASPAR `Arnt.sites` file showing known binding sites of the mouse helix-loop-helix transcription factor Arnt.

```
>MA0004 ARNT 1
CACGTGatgtcctc
>MA0004 ARNT 2
CACGTGggaggtac
>MA0004 ARNT 3
CACGTGccgcgcgc
...
>MA0004 ARNT 18
AACGTGacagccctcc
>MA0004 ARNT 19
AACGTGcacatcgtcc
>MA0004 ARNT 20
aggaatCGCGTGc
```

The parts of the sequence in capital letters are the motif instances that were found to align to each other.

We can create a `Motif` object from these instances as follows:

```
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     arnt = motifs.read(handle, "sites")
...
```

The instances from which this motif was created is stored in the `.alignment` property:

```
>>> print(arnt.alignment.sequences[:3])
[Seq('CACGTG'), Seq('CACGTG'), Seq('CACGTG')]
>>> for sequence in arnt.alignment.sequences:
...     print(sequence)
...
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
AACGTG
AACGTG
AACGTG
AACGTG
CGCGTG
```

The counts matrix of this motif is automatically calculated from the instances:

```
>>> print(arnt.counts)
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
<BLANKLINE>
```

This format does not store any meta information.

The JASPAR `pfm` format

JASPAR also makes motifs available directly as a count matrix, without the instances from which it was created. This `pfm` format only stores the counts matrix for a single motif. For example, this is the JASPAR file `SRF.pfm` containing the counts matrix for the human SRF transcription factor:

```
 2 9 0 1 32 3 46 1 43 15 2 2
 1 33 45 45 1 1 0 0 0 1 0 1
39 2 1 0 0 0 0 0 0 0 44 43
 4 2 0 0 13 42 0 45 3 30 0 0
```

We can create a motif for this count matrix as follows:

```
>>> with open("SRF.pfm") as handle:
...     srf = motifs.read(handle, "pfm")
...
>>> print(srf.counts)
        0      1      2      3      4      5      6      7      8      9     10     11
A:   2.00   9.00   0.00   1.00  32.00   3.00  46.00   1.00  43.00  15.00   2.00   2.00
C:   1.00  33.00  45.00  45.00   1.00   1.00   0.00   0.00   0.00   1.00   0.00   1.00
G:  39.00   2.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00  44.00  43.00
T:   4.00   2.00   0.00   0.00  13.00  42.00   0.00  45.00   3.00  30.00   0.00   0.00
<BLANKLINE>
```

As this motif was created from the counts matrix directly, it has no instances associated with it:

```
>>> print(srf.alignment)
None
```

We can now ask for the consensus sequence of these two motifs:

```
>>> print(arnt.counts.consensus)
CACGTG
>>> print(srf.counts.consensus)
GCCCATATATGG
```

As with the instances file, no meta information is stored in this format.

The JASPAR format `jaspar`

The `jaspar` file format allows multiple motifs to be specified in a single file. In this format each of the motif records consist of a header line followed by four lines defining the counts matrix. The header line begins with a `>` character (similar to the Fasta file format) and is followed by the unique JASPAR matrix ID and the TF name. The following example shows a `jaspar` formatted file containing the three motifs Arnt, RUNX1 and MEF2A:

```
>MA0004.1 Arnt
A  [ 4 19  0  0  0  0 ]
C  [16  0 20  0  0  0 ]
G  [ 0  1  0 20  0 20 ]
T  [ 0  0  0  0 20  0 ]
>MA0002.1 RUNX1
A  [10 12  4  1  2  2  0  0  0  8 13 ]
C  [ 2  2  7  1  0  8  0  0  1  2  2 ]
G  [ 3  1  1  0 23  0 26 26  0  0  4 ]
T  [11 11 14 24  1 16  0  0 25 16  7 ]
>MA0052.1 MEF2A
A  [ 1  0 57  2  9  6 37  2 56  6 ]
C  [50  0  1  1  0  0  0  0  0  0 ]
G  [ 0  0  0  0  0  0  0  0  2 50 ]
T  [ 7 58  0 55 49 52 21 56  0  2 ]
```

The motifs are read as follows:

```
>>> fh = open("jaspar_motifs.txt")
>>> for m in motifs.parse(fh, "jaspar"):
...     print(m)
...
TF name         Arnt
Matrix ID        MA0004.1
Matrix:
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00



TF name         RUNX1
Matrix ID        MA0002.1
Matrix:
        0      1      2      3      4      5      6      7      8      9     10
A:  10.00  12.00   4.00   1.00   2.00   2.00   0.00   0.00   0.00   8.00  13.00
C:   2.00   2.00   7.00   1.00   0.00   8.00   0.00   0.00   1.00   2.00   2.00
G:   3.00   1.00   1.00   0.00  23.00   0.00  26.00  26.00   0.00   0.00   4.00
T:  11.00  11.00  14.00  24.00   1.00  16.00   0.00   0.00  25.00  16.00   7.00



TF name         MEF2A
Matrix ID        MA0052.1
Matrix:
        0      1      2      3      4      5      6      7      8      9
A:   1.00   0.00  57.00   2.00   9.00   6.00  37.00   2.00  56.00   6.00
C:  50.00   0.00   1.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00
G:   0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   2.00  50.00
T:   7.00  58.00   0.00  55.00  49.00  52.00  21.00  56.00   0.00   2.00
```

Note that printing a JASPAR motif yields both the counts data and the available meta-information.

Accessing the JASPAR database

In addition to parsing these flat file formats, we can also retrieve motifs from a JASPAR SQL database. Unlike the flat file formats, a JASPAR database allows storing of all possible meta information defined in the JASPAR `Motif` class. It is beyond the scope of this document to describe how to set up a JASPAR database (please see the main JASPAR website). Motifs are read from a JASPAR database using the `Bio.motifs.jaspar.db` module. First connect to the JASPAR database using the JASPAR5 class which models the the latest JASPAR schema:

```
>>> from Bio.motifs.jaspar.db import JASPAR5
>>>
>>> JASPAR_DB_HOST = "yourhostname"  # fill in these values
>>> JASPAR_DB_NAME = "yourdatabase"
>>> JASPAR_DB_USER = "yourusername"
```

```
>>> JASPAR_DB_PASS = "yourpassword"
>>>
>>> jdb = JASPAR5(
...     host=JASPAR_DB_HOST,
...     name=JASPAR_DB_NAME,
...     user=JASPAR_DB_USER,
...     password=JASPAR_DB_PASS,
... )
```

Now we can fetch a single motif by its unique JASPAR ID with the `fetch_motif_by_id` method. Note that a JASPAR ID consists of a base ID and a version number separated by a decimal point, e.g. 'MA0004.1'. The `fetch_motif_by_id` method allows you to use either the fully specified ID or just the base ID. If only the base ID is provided, the latest version of the motif is returned.

```
>>> arnt = jdb.fetch_motif_by_id("MA0004")
```

Printing the motif reveals that the JASPAR SQL database stores much more meta-information than the flat files:

```
>>> print(arnt)
TF name         Arnt
Matrix ID         MA0004.1
Collection         CORE
TF class         Zipper-Type
TF family         Helix-Loop-Helix
Species         10090
Taxonomic group         vertebrates
Accession         ['P53762']
Data type used         SELEX
Medline         7592839
PAZAR ID         TF0000003
Comments         -
Matrix:
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
```

We can also fetch motifs by name. The name must be an exact match (partial matches or database wildcards are not currently supported). Note that as the name is not guaranteed to be unique, the `fetch_motifs_by_name` method actually returns a list.

```
>>> motifs = jdb.fetch_motifs_by_name("Arnt")
>>> print(motifs[0])
TF name         Arnt
Matrix ID         MA0004.1
Collection         CORE
TF class         Zipper-Type
TF family         Helix-Loop-Helix
Species         10090
Taxonomic group         vertebrates
```

```
Accession        ['P53762']
Data type used        SELEX
Medline         7592839
PAZAR ID         TF0000003
Comments         -
Matrix:
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
```

The `fetch_motifs` method allows you to fetch motifs which match a specified set of criteria. These criteria include any of the above described meta information as well as certain matrix properties such as the minimum information content (`min_ic` in the example below), the minimum length of the matrix or the minimum number of sites used to construct the matrix. Only motifs which pass ALL the specified criteria are returned. Note that selection criteria which correspond to meta information which allow for multiple values may be specified as either a single value or a list of values, e.g. `tax_group` and `tf_family` in the example below.

```
>>> motifs = jdb.fetch_motifs(
...     collection="CORE",
...     tax_group=["vertebrates", "insects"],
...     tf_class="Winged Helix-Turn-Helix",
...     tf_family=["Forkhead", "Ets"],
...     min_ic=12,
... )
>>> for motif in motifs:
...     pass  # do something with the motif
...
```

Compatibility with Perl TFBS modules

An important thing to note is that the JASPAR `Motif` class was designed to be compatible with the popular Perl TFBS modules. Therefore some specifics about the choice of defaults for background and pseudocounts as well as how information content is computed and sequences searched for instances is based on this compatibility criteria. These choices are noted in the specific subsections below.

- **Choice of background:**
  The Perl `TFBS` modules appear to allow a choice of custom background probabilities (although the documentation states that uniform background is assumed). However the default is to use a uniform background. Therefore it is recommended that you use a uniform background for computing the position-specific scoring matrix (PSSM). This is the default when using the Biopython `motifs` module.

- **Choice of pseudocounts:**
  By default, the Perl `TFBS` modules use a pseudocount equal to $\sqrt{N} * \text{bg[nucleotide]}$, where N represents the total number of sequences used to construct the matrix. To apply this same pseudocount formula, set the motif `pseudocounts` attribute using the `jaspar.calculate\_pseudcounts()` function:

  ```
  >>> motif.pseudocounts = motifs.jaspar.calculate_pseudocounts(motif)
  ```

Note that it is possible for the counts matrix to have an unequal number of sequences making up the columns. The pseudocount computation uses the average number of sequences making up the matrix. However, when `normalize` is called on the counts matrix, each count value in a column is divided by the total number of sequences making up that specific column, not by the average number of sequences. This differs from the Perl `TFBS` modules because the normalization is not done as a separate step and so the average number of sequences is used throughout the computation of the pssm. Therefore, for matrices with unequal column counts, the PSSM computed by the `motifs` module will differ somewhat from the pssm computed by the Perl `TFBS` modules.

- **Computation of matrix information content:**
  The information content (IC) or specificity of a matrix is computed using the `mean` method of the `PositionSpecificScoringMatrix` class. However of note, in the Perl `TFBS` modules the default behavior is to compute the IC without first applying pseudocounts, even though by default the PSSMs are computed using pseudocounts as described above.

- **Searching for instances:**
  Searching for instances with the Perl `TFBS` motifs was usually performed using a relative score threshold, i.e. a score in the range 0 to 1. In order to compute the absolute PSSM score corresponding to a relative score one can use the equation:

```
>>> abs_score = (pssm.max - pssm.min) * rel_score + pssm.min
```

  To convert the absolute score of an instance back to a relative score, one can use the equation:

```
>>> rel_score = (abs_score - pssm.min) / (pssm.max - pssm.min)
```

  For example, using the Arnt motif before, let's search a sequence with a relative score threshold of 0.8.

```
>>> test_seq = Seq("TAAGCGTGCACGCGCAACACGTGCATTA")
>>> arnt.pseudocounts = motifs.jaspar.calculate_pseudocounts(arnt)
>>> pssm = arnt.pssm
>>> max_score = pssm.max
>>> min_score = pssm.min
>>> abs_score_threshold = (max_score - min_score) * 0.8 + min_score
>>> for pos, score in pssm.search(test_seq, threshold=abs_score_threshold):
...     rel_score = (score - min_score) / (max_score - min_score)
...     print(f"Position {pos}: score = {score:5.3f}, rel. score = {rel_score:5.3f}")
...
Position 2: score = 5.362, rel. score = 0.801
Position 8: score = 6.112, rel. score = 0.831
Position -20: score = 7.103, rel. score = 0.870
Position 17: score = 10.351, rel. score = 1.000
Position -11: score = 10.351, rel. score = 1.000
```

17.2.2 MEME

MEME [2] is a tool for discovering motifs in a group of related DNA or protein sequences. It takes as input a group of DNA or protein sequences and outputs as many motifs as requested. Therefore, in contrast to JASPAR files, MEME output files typically contain multiple motifs. This is an example.

At the top of an output file generated by MEME shows some background information about the MEME and the version of MEME used:

```
********************************************************************************
MEME - Motif discovery tool
********************************************************************************
```

```
MEME version 3.0 (Release date: 2004/08/18 09:07:01)
...
```

Further down, the input set of training sequences is recapitulated:

```
********************************************************************************
TRAINING SET
********************************************************************************
DATAFILE= INO_up800.s
ALPHABET= ACGT
Sequence name            Weight Length  Sequence name            Weight Length
-------------            ------ ------  -------------            ------ ------
CHO1                     1.0000    800  CHO2                     1.0000    800
FAS1                     1.0000    800  FAS2                     1.0000    800
ACC1                     1.0000    800  INO1                     1.0000    800
OPI3                     1.0000    800
********************************************************************************
```

and the exact command line that was used:

```
********************************************************************************
COMMAND LINE SUMMARY
********************************************************************************
This information can also be useful in the event you wish to report a
problem with the MEME software.

command: meme -mod oops -dna -revcomp -nmotifs 2 -bfile yeast.nc.6.freq INO_up800.s
...
```

Next is detailed information on each motif that was found:

```
********************************************************************************
MOTIF  1        width =   12   sites =   7   llr = 95   E-value = 2.0e-001
********************************************************************************
--------------------------------------------------------------------------------
        Motif 1 Description
--------------------------------------------------------------------------------
Simplified        A  :::9:a::::3:
pos.-specific     C  ::a:9:11691a
probability       G  ::::1::94:4:
matrix            T  aa:1::9::11:
```

To parse this file (stored as `meme.dna.oops.txt`), use

```
>>> with open("meme.INO_up800.classic.oops.xml") as handle:
...     record = motifs.parse(handle, "meme")
...
```

The `motifs.parse` command reads the complete file directly, so you can close the file after calling `motifs.parse`. The header information is stored in attributes:

```
>>> record.version
'5.0.1'
>>> record.datafile
'common/INO_up800.s'
```

```
>>> record.command
'meme common/INO_up800.s -oc results/meme10 -mod oops -dna -revcomp -bfile common/yeast.nc.6.freq -nmoti
>>> record.alphabet
'ACGT'
>>> record.sequences
['sequence_0', 'sequence_1', 'sequence_2', 'sequence_3', 'sequence_4', 'sequence_5', 'sequence_6']
```

The record is an object of the `Bio.motifs.meme.Record` class. The class inherits from list, and you can think of `record` as a list of Motif objects:

```
>>> len(record)
2
>>> motif = record[0]
>>> print(motif.consensus)
GCGGCATGTGAAA
>>> print(motif.degenerate_consensus)
GSKGCATGTGAAA
```

In addition to these generic motif attributes, each motif also stores its specific information as calculated by MEME. For example,

```
>>> motif.num_occurrences
7
>>> motif.length
13
>>> evalue = motif.evalue
>>> print("%3.1g" % evalue)
0.2
>>> motif.name
'GSKGCATGTGAAA'
>>> motif.id
'motif_1'
```

In addition to using an index into the record, as we did above, you can also find it by its name:

```
>>> motif = record["GSKGCATGTGAAA"]
```

Each motif has an attribute `.alignment` with the sequence alignment in which the motif was found, providing some information on each of the sequences:

```
>>> len(motif.alignment)
7
>>> motif.alignment.sequences[0]
Instance('GCGGCATGTGAAA')
>>> motif.alignment.sequences[0].motif_name
'GSKGCATGTGAAA'
>>> motif.alignment.sequences[0].sequence_name
'INO1'
>>> motif.alignment.sequences[0].sequence_id
'sequence_5'
>>> motif.alignment.sequences[0].start
620
>>> motif.alignment.sequences[0].strand
```

```
'+'
>>> motif.alignment.sequences[0].length
13
>>> pvalue = motif.alignment.sequences[0].pvalue
>>> print("%5.3g" % pvalue)
1.21e-08
```

MAST

17.2.3 TRANSFAC

TRANSFAC is a manually curated database of transcription factors, together with their genomic binding sites and DNA binding profiles [32]. While the file format used in the TRANSFAC database is nowadays also used by others, we will refer to it as the TRANSFAC file format.

A minimal file in the TRANSFAC format looks as follows:

```
ID  motif1
P0      A       C       G       T
01      1       2       2       0       S
02      2       1       2       0       R
03      3       0       1       1       A
04      0       5       0       0       C
05      5       0       0       0       A
06      0       0       4       1       G
07      0       1       4       0       G
08      0       0       0       5       T
09      0       0       5       0       G
10      0       1       2       2       K
11      0       2       0       3       Y
12      1       0       3       1       G
//
```

This file shows the frequency matrix of motif `motif1` of 12 nucleotides. In general, one file in the TRANSFAC format can contain multiple motifs. For example, this is the contents of the example TRANSFAC file `transfac.dat`:

```
VV  EXAMPLE January 15, 2013
XX
//
ID  motif1
P0      A       C       G       T
01      1       2       2       0       S
02      2       1       2       0       R
03      3       0       1       1       A
...
11      0       2       0       3       Y
12      1       0       3       1       G
//
ID  motif2
P0      A       C       G       T
01      2       1       2       0       R
02      1       2       2       0       S
...
```

```
09      0      0      0      5      T
10      0      2      0      3      Y
//
```

To parse a TRANSFAC file, use

```
>>> with open("transfac.dat") as handle:
...     record = motifs.parse(handle, "TRANSFAC")
...
```

If any discrepancies between the file contents and the TRANSFAC file format are detected, a `ValueError` is raised. Note that you may encounter files that do not follow the TRANSFAC format strictly. For example, the number of spaces between columns may be different, or a tab may be used instead of spaces. Use `strict=False` to enable parsing such files without raising a `ValueError`:

```
>>> record = motifs.parse(handle, "TRANSFAC", strict=False)
```

When parsing a non-compliant file, we recommend to check the record returned by `motif.parse` to ensure that it is consistent with the file contents.

The overall version number, if available, is stored as `record.version`:

```
>>> record.version
'EXAMPLE January 15, 2013'
```

Each motif in `record` is in instance of the `Bio.motifs.transfac.Motif` class, which inherits both from the `Bio.motifs.Motif` class and from a Python dictionary. The dictionary uses the two-letter keys to store any additional information about the motif:

```
>>> motif = record[0]
>>> motif.degenerate_consensus  # Using the Bio.motifs.Motif property
Seq('SRACAGGTGKYG')
>>> motif["ID"]  # Using motif as a dictionary
'motif1'
```

TRANSFAC files are typically much more elaborate than this example, containing lots of additional information about the motif. Table 17.2.3 lists the two-letter field codes that are commonly found in TRANSFAC files:

Each motif also has an attribute `.references` containing the references associated with the motif, using these two-letter keys:

Printing the motifs writes them out in their native TRANSFAC format:

```
>>> print(record)
VV  EXAMPLE January 15, 2013
XX
//
ID  motif1
XX
P0      A      C      G      T
01      1      2      2      0      S
02      2      1      2      0      R
03      3      0      1      1      A
04      0      5      0      0      C
05      5      0      0      0      A
06      0      0      4      1      G
```

Table 17.1: Fields commonly found in TRANSFAC files

AC	Accession number
AS	Accession numbers, secondary
BA	Statistical basis
BF	Binding factors
BS	Factor binding sites underlying the matrix
CC	Comments
CO	Copyright notice
DE	Short factor description
DR	External databases
DT	Date created/updated
HC	Subfamilies
HP	Superfamilies
ID	Identifier
NA	Name of the binding factor
OC	Taxonomic classification
OS	Species/Taxon
OV	Older version
PV	Preferred version
TY	Type
XX	Empty line; these are not stored in the Record.

Table 17.2: Fields used to store references in TRANSFAC files

RN	Reference number
RA	Reference authors
RL	Reference data
RT	Reference title
RX	PubMed ID

```
07      0       1       4       0       G
08      0       0       0       5       T
09      0       0       5       0       G
10      0       1       2       2       K
11      0       2       0       3       Y
12      1       0       3       1       G
XX
//
ID  motif2
XX
P0      A       C       G       T
01      2       1       2       0       R
02      1       2       2       0       S
03      0       5       0       0       C
04      3       0       1       1       A
05      0       0       4       1       G
06      5       0       0       0       A
07      0       1       4       0       G
08      0       0       5       0       G
```

```
09      0      0      0      5      T
10      0      2      0      3      Y
XX
//
<BLANKLINE>
```

You can export the motifs in the TRANSFAC format by capturing this output in a string and saving it in a file:

```
>>> text = str(record)
>>> with open("mytransfacfile.dat", "w") as out_handle:
...     out_handle.write(text)
...
```

17.3 Writing motifs

Speaking of exporting, let's look at export functions in general. We can use the `format` built-in function to write the motif in the simple JASPAR `pfm` format:

```
>>> print(format(arnt, "pfm"))
  4.00  19.00   0.00   0.00   0.00   0.00
 16.00   0.00  20.00   0.00   0.00   0.00
  0.00   1.00   0.00  20.00   0.00  20.00
  0.00   0.00   0.00   0.00  20.00   0.00
```

Similarly, we can use `format` to write the motif in the JASPAR `jaspar` format:

```
>>> print(format(arnt, "jaspar"))
>MA0004.1  Arnt
A [  4.00  19.00   0.00   0.00   0.00   0.00]
C [ 16.00   0.00  20.00   0.00   0.00   0.00]
G [  0.00   1.00   0.00  20.00   0.00  20.00]
T [  0.00   0.00   0.00   0.00  20.00   0.00]
```

To write the motif in a TRANSFAC-like matrix format, use

```
>>> print(format(m, "transfac"))
PO      A      C      G      T
01      3      0      0      4      W
02      7      0      0      0      A
03      0      5      0      2      C
04      2      2      3      0      V
05      1      6      0      0      C
XX
//
<BLANKLINE>
```

To write out multiple motifs, you can use `motifs.write`. This function can be used regardless of whether the motifs originated from a TRANSFAC file. For example,

```
>>> two_motifs = [arnt, srf]
>>> print(motifs.write(two_motifs, "transfac"))
PO      A      C      G      T
01      4     16      0      0      C
```

```
02     19      0      1      0      A
03      0     20      0      0      C
04      0      0     20      0      G
05      0      0      0     20      T
06      0      0     20      0      G
XX
//
PO      A      C      G      T
01      2      1     39      4      G
02      9     33      2      2      C
03      0     45      1      0      C
04      1     45      0      0      C
05     32      1      0     13      A
06      3      1      0     42      T
07     46      0      0      0      A
08      1      0      0     45      T
09     43      0      0      3      A
10     15      1      0     30      W
11      2      0     44      0      G
12      2      1     43      0      G
XX
//
<BLANKLINE>
```

Or, to write multiple motifs in the `jaspar` format:

```
>>> two_motifs = [arnt, mef2a]
>>> print(motifs.write(two_motifs, "jaspar"))
>MA0004.1  Arnt
A [  4.00  19.00   0.00   0.00   0.00   0.00]
C [ 16.00   0.00  20.00   0.00   0.00   0.00]
G [  0.00   1.00   0.00  20.00   0.00  20.00]
T [  0.00   0.00   0.00   0.00  20.00   0.00]
>MA0052.1  MEF2A
A [  1.00   0.00  57.00   2.00   9.00   6.00  37.00   2.00  56.00   6.00]
C [ 50.00   0.00   1.00   1.00   0.00   0.00   0.00   0.00   0.00   0.00]
G [  0.00   0.00   0.00   0.00   0.00   0.00   0.00   0.00   2.00  50.00]
T [  7.00  58.00   0.00  55.00  49.00  52.00  21.00  56.00   0.00   2.00]
```

17.4 Position-Weight Matrices

The `.counts` attribute of a Motif object shows how often each nucleotide appeared at each position along the alignment. We can normalize this matrix by dividing by the number of instances in the alignment, resulting in the probability of each nucleotide at each position along the alignment. We refer to these probabilities as the position-weight matrix. However, beware that in the literature this term may also be used to refer to the position-specific scoring matrix, which we discuss below.

Usually, pseudocounts are added to each position before normalizing. This avoids overfitting of the position-weight matrix to the limited number of motif instances in the alignment, and can also prevent probabilities from becoming zero. To add a fixed pseudocount to all nucleotides at all positions, specify a number for the `pseudocounts` argument:

```
>>> pwm = m.counts.normalize(pseudocounts=0.5)
>>> print(pwm)
```

```
        0      1      2      3      4
A:   0.39   0.83   0.06   0.28   0.17
C:   0.06   0.06   0.61   0.28   0.72
G:   0.06   0.06   0.06   0.39   0.06
T:   0.50   0.06   0.28   0.06   0.06
<BLANKLINE>
```

Alternatively, `pseudocounts` can be a dictionary specifying the pseudocounts for each nucleotide. For example, as the GC content of the human genome is about 40%, you may want to choose the pseudocounts accordingly:

```
>>> pwm = m.counts.normalize(pseudocounts={"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6})
>>> print(pwm)
        0      1      2      3      4
A:   0.40   0.84   0.07   0.29   0.18
C:   0.04   0.04   0.60   0.27   0.71
G:   0.04   0.04   0.04   0.38   0.04
T:   0.51   0.07   0.29   0.07   0.07
<BLANKLINE>
```

The position-weight matrix has its own methods to calculate the consensus, anticonsensus, and degenerate consensus sequences:

```
>>> pwm.consensus
Seq('TACGC')
>>> pwm.anticonsensus
Seq('CCGTG')
>>> pwm.degenerate_consensus
Seq('WACNC')
```

Note that due to the pseudocounts, the degenerate consensus sequence calculated from the position-weight matrix is slightly different from the degenerate consensus sequence calculated from the instances in the motif:

```
>>> m.degenerate_consensus
Seq('WACVC')
```

The reverse complement of the position-weight matrix can be calculated directly from the `pwm`:

```
>>> rpwm = pwm.reverse_complement()
>>> print(rpwm)
        0      1      2      3      4
A:   0.07   0.07   0.29   0.07   0.51
C:   0.04   0.38   0.04   0.04   0.04
G:   0.71   0.27   0.60   0.04   0.04
T:   0.18   0.29   0.07   0.84   0.40
<BLANKLINE>
```

17.5 Position-Specific Scoring Matrices

Using the background distribution and PWM with pseudo-counts added, it's easy to compute the log-odds ratios, telling us what are the log odds of a particular symbol to be coming from a motif against the background. We can use the `.log_odds()` method on the position-weight matrix:

```
>>> pssm = pwm.log_odds()
>>> print(pssm)
        0      1      2      3      4
A:   0.68   1.76  -1.91   0.21  -0.49
C:  -2.49  -2.49   1.26   0.09   1.51
G:  -2.49  -2.49  -2.49   0.60  -2.49
T:   1.03  -1.91   0.21  -1.91  -1.91
<BLANKLINE>
```

Here we can see positive values for symbols more frequent in the motif than in the background and negative for symbols more frequent in the background. 0.0 means that it's equally likely to see a symbol in the background and in the motif.

This assumes that A, C, G, and T are equally likely in the background. To calculate the position-specific scoring matrix against a background with unequal probabilities for A, C, G, T, use the `background` argument. For example, against a background with a 40% GC content, use

```
>>> background = {"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3}
>>> pssm = pwm.log_odds(background)
>>> print(pssm)
        0      1      2      3      4
A:   0.42   1.49  -2.17  -0.05  -0.75
C:  -2.17  -2.17   1.58   0.42   1.83
G:  -2.17  -2.17  -2.17   0.92  -2.17
T:   0.77  -2.17  -0.05  -2.17  -2.17
<BLANKLINE>
```

The maximum and minimum score obtainable from the PSSM are stored in the `.max` and `.min` properties:

```
>>> print("%4.2f" % pssm.max)
6.59
>>> print("%4.2f" % pssm.min)
-10.85
```

The mean and standard deviation of the PSSM scores with respect to a specific background are calculated by the `.mean` and `.std` methods.

```
>>> mean = pssm.mean(background)
>>> std = pssm.std(background)
>>> print("mean = %0.2f, standard deviation = %0.2f" % (mean, std))
mean = 3.21, standard deviation = 2.59
```

A uniform background is used if `background` is not specified. The mean is equal to the Kullback-Leibler divergence or relative entropy described in Section 17.1.5.

The `.reverse_complement`, `.consensus`, `.anticonsensus`, and `.degenerate_consensus` methods can be applied directly to PSSM objects.

17.6 Searching for instances

The most frequent use for a motif is to find its instances in some sequence. For the sake of this section, we will use an artificial sequence like this:

```
>>> test_seq = Seq("TACACTGCATTACAACCCAAGCATTA")
>>> len(test_seq)
26
```

17.6.1 Searching for exact matches

The simplest way to find instances, is to look for exact matches of the true instances of the motif:

```
>>> for pos, seq in test_seq.search(m.alignment):
...     print("%i %s" % (pos, seq))
...
0 TACAC
10 TACAA
13 AACCC
```

We can do the same with the reverse complement (to find instances on the complementary strand):

```
>>> for pos, seq in test_seq.search(r.alignment):
...     print("%i %s" % (pos, seq))
...
6 GCATT
20 GCATT
```

17.6.2 Searching for matches using the PSSM score

It's just as easy to look for positions, giving rise to high log-odds scores against our motif:

```
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 10: score = 3.037
Position 13: score = 5.738
Position -6: score = 4.601
```

The negative positions refer to instances of the motif found on the reverse strand of the test sequence, and follow the Python convention on negative indices. Therefore, the instance of the motif at `pos` is located at `test_seq[pos:pos+len(m)]` both for positive and for negative values of `pos`.

You may notice the threshold parameter, here set arbitrarily to 3.0. This is in log_2, so we are now looking only for words, which are eight times more likely to occur under the motif model than in the background. The default threshold is 0.0, which selects everything that looks more like the motif than the background.

You can also calculate the scores at all positions along the sequence:

```
>>> pssm.calculate(test_seq)
array([  5.62230396,  -5.6796999 ,  -3.43177247,   0.93827754,
        -6.84962511,  -2.04066086, -10.84962463,  -3.65614533,
        -0.03370807,  -3.91102552,   3.03734159,  -2.14918518,
        -0.6016975 ,   5.7381525 ,  -0.50977498,  -3.56422281,
        -8.73414803,  -0.09919716,  -0.6016975 ,  -2.39429784,
       -10.84962463,  -3.65614533], dtype=float32)
```

In general, this is the fastest way to calculate PSSM scores. The scores returned by `pssm.calculate` are for the forward strand only. To obtain the scores on the reverse strand, you can take the reverse complement of the PSSM:

```
>>> rpssm = pssm.reverse_complement()
>>> rpssm.calculate(test_seq)
```

```
array([ -9.43458748,  -3.06172252,  -7.18665981,  -7.76216221,
        -2.04066086,  -4.26466274,   4.60124254,  -4.2480607 ,
        -8.73414803,  -2.26503372,  -6.49598789,  -5.64668512,
        -8.73414803, -10.84962463,  -4.82356262,  -4.82356262,
        -5.64668512,  -8.73414803,  -4.15613794,  -5.6796999 ,
         4.60124254,  -4.2480607 ], dtype=float32)
```

17.6.3 Selecting a score threshold

If you want to use a less arbitrary way of selecting thresholds, you can explore the distribution of PSSM scores. Since the space for a score distribution grows exponentially with motif length, we are using an approximation with a given precision to keep computation cost manageable:

```
>>> distribution = pssm.distribution(background=background, precision=10**4)
```

The `distribution` object can be used to determine a number of different thresholds. We can specify the requested false-positive rate (probability of "finding" a motif instance in background generated sequence):

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
```

or the false-negative rate (probability of "not finding" an instance generated from the motif):

```
>>> threshold = distribution.threshold_fnr(0.1)
>>> print("%5.3f" % threshold)
-0.510
```

or a threshold (approximately) satisfying some relation between the false-positive rate and the false-negative rate ($\frac{\text{fnr}}{\text{fpr}} \simeq t$):

```
>>> threshold = distribution.threshold_balanced(1000)
>>> print("%5.3f" % threshold)
6.241
```

or a threshold satisfying (roughly) the equality between the $-log$ of the false-positive rate and the information content (as used in patser software by Hertz and Stormo):

```
>>> threshold = distribution.threshold_patser()
>>> print("%5.3f" % threshold)
0.346
```

For example, in case of our motif, you can get the threshold giving you exactly the same results (for this sequence) as searching for instances with balanced threshold with rate of 1000.

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%5.3f" % threshold)
4.009
>>> for position, score in pssm.search(test_seq, threshold=threshold):
...     print("Position %d: score = %5.3f" % (position, score))
...
Position 0: score = 5.622
Position -20: score = 4.601
Position 13: score = 5.738
Position -6: score = 4.601
```

17.7 Each motif object has an associated Position-Specific Scoring Matrix

To facilitate searching for potential TFBSs using PSSMs, both the position-weight matrix and the position-specific scoring matrix are associated with each motif. Using the Arnt motif as an example:

```
>>> from Bio import motifs
>>> with open("Arnt.sites") as handle:
...     motif = motifs.read(handle, "sites")
...
>>> print(motif.counts)
        0      1      2      3      4      5
A:   4.00  19.00   0.00   0.00   0.00   0.00
C:  16.00   0.00  20.00   0.00   0.00   0.00
G:   0.00   1.00   0.00  20.00   0.00  20.00
T:   0.00   0.00   0.00   0.00  20.00   0.00
<BLANKLINE>
>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.20   0.95   0.00   0.00   0.00   0.00
C:   0.80   0.00   1.00   0.00   0.00   0.00
G:   0.00   0.05   0.00   1.00   0.00   1.00
T:   0.00   0.00   0.00   0.00   1.00   0.00
<BLANKLINE>
>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.32   1.93   -inf   -inf   -inf   -inf
C:   1.68   -inf   2.00   -inf   -inf   -inf
G:   -inf  -2.32   -inf   2.00   -inf   2.00
T:   -inf   -inf   -inf   -inf   2.00   -inf
<BLANKLINE>
```

The negative infinities appear here because the corresponding entry in the frequency matrix is 0, and we are using zero pseudocounts by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 0.00
C: 0.00
G: 0.00
T: 0.00
```

If you change the `.pseudocounts` attribute, the position-frequency matrix and the position-specific scoring matrix are recalculated automatically:

```
>>> motif.pseudocounts = 3.0
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.pseudocounts[letter]))
...
A: 3.00
C: 3.00
```

```
G: 3.00
T: 3.00
>>> print(motif.pwm)
        0      1      2      3      4      5
A:   0.22   0.69   0.09   0.09   0.09   0.09
C:   0.59   0.09   0.72   0.09   0.09   0.09
G:   0.09   0.12   0.09   0.72   0.09   0.72
T:   0.09   0.09   0.09   0.09   0.72   0.09
<BLANKLINE>

>>> print(motif.pssm)
        0      1      2      3      4      5
A:  -0.19   1.46  -1.42  -1.42  -1.42  -1.42
C:   1.25  -1.42   1.52  -1.42  -1.42  -1.42
G:  -1.42  -1.00  -1.42   1.52  -1.42   1.52
T:  -1.42  -1.42  -1.42  -1.42   1.52  -1.42
<BLANKLINE>
```

You can also set the `.pseudocounts` to a dictionary over the four nucleotides if you want to use different pseudocounts for them. Setting `motif.pseudocounts` to `None` resets it to its default value of zero.

The position-specific scoring matrix depends on the background distribution, which is uniform by default:

```
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

Again, if you modify the background distribution, the position-specific scoring matrix is recalculated:

```
>>> motif.background = {"A": 0.2, "C": 0.3, "G": 0.3, "T": 0.2}
>>> print(motif.pssm)
        0      1      2      3      4      5
A:   0.13   1.78  -1.09  -1.09  -1.09  -1.09
C:   0.98  -1.68   1.26  -1.68  -1.68  -1.68
G:  -1.68  -1.26  -1.68   1.26  -1.68   1.26
T:  -1.09  -1.09  -1.09  -1.09   1.85  -1.09
<BLANKLINE>
```

Setting `motif.background` to `None` resets it to a uniform distribution:

```
>>> motif.background = None
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

If you set `motif.background` equal to a single value, it will be interpreted as the GC content:

```
>>> motif.background = 0.8
>>> for letter in "ACGT":
...     print("%s: %4.2f" % (letter, motif.background[letter]))
...
A: 0.10
C: 0.40
G: 0.40
T: 0.10
```

Note that you can now calculate the mean of the PSSM scores over the background against which it was computed:

```
>>> print("%f" % motif.pssm.mean(motif.background))
4.703928
```

as well as its standard deviation:

```
>>> print("%f" % motif.pssm.std(motif.background))
3.290900
```

and its distribution:

```
>>> distribution = motif.pssm.distribution(background=motif.background)
>>> threshold = distribution.threshold_fpr(0.01)
>>> print("%f" % threshold)
3.854375
```

Note that the position-weight matrix and the position-specific scoring matrix are recalculated each time you call `motif.pwm` or `motif.pssm`, respectively. If speed is an issue and you want to use the PWM or PSSM repeatedly, you can save them as a variable, as in

```
>>> pssm = motif.pssm
```

17.8 Comparing motifs

Once we have more than one motif, we might want to compare them.

Before we start comparing motifs, I should point out that motif boundaries are usually quite arbitrary. This means we often need to compare motifs of different lengths, so comparison needs to involve some kind of alignment. This means we have to take into account two things:

- alignment of motifs
- some function to compare aligned motifs

To align the motifs, we use ungapped alignment of PSSMs and substitute zeros for any missing columns at the beginning and end of the matrices. This means that effectively we are using the background distribution for columns missing from the PSSM. The distance function then returns the minimal distance between motifs, as well as the corresponding offset in their alignment.

To give an example, let us first load another motif, which is similar to our test motif `m`:

```
>>> with open("REB1.pfm") as handle:
...     m_reb1 = motifs.read(handle, "pfm")
...
>>> m_reb1.consensus
```

```
Seq('GTTACCCGG')
>>> print(m_reb1.counts)
        0      1      2      3      4      5      6      7      8
A:  30.00   0.00   0.00 100.00   0.00   0.00   0.00   0.00  15.00
C:  10.00   0.00   0.00   0.00 100.00 100.00 100.00   0.00  15.00
G:  50.00   0.00   0.00   0.00   0.00   0.00   0.00  60.00  55.00
T:  10.00 100.00 100.00   0.00   0.00   0.00   0.00  40.00  15.00
<BLANKLINE>
```

To make the motifs comparable, we choose the same values for the pseudocounts and the background distribution as our motif `m`:

```
>>> m_reb1.pseudocounts = {"A": 0.6, "C": 0.4, "G": 0.4, "T": 0.6}
>>> m_reb1.background = {"A": 0.3, "C": 0.2, "G": 0.2, "T": 0.3}
>>> pssm_reb1 = m_reb1.pssm
>>> print(pssm_reb1)
        0      1      2      3      4      5      6      7      8
A:   0.00  -5.67  -5.67   1.72  -5.67  -5.67  -5.67  -5.67  -0.97
C:  -0.97  -5.67  -5.67  -5.67   2.30   2.30   2.30  -5.67  -0.41
G:   1.30  -5.67  -5.67  -5.67  -5.67  -5.67  -5.67   1.57   1.44
T:  -1.53   1.72   1.72  -5.67  -5.67  -5.67  -5.67   0.41  -0.97
<BLANKLINE>
```

We'll compare these motifs using the Pearson correlation. Since we want it to resemble a distance measure, we actually take $1 - r$, where r is the Pearson correlation coefficient (PCC):

```
>>> distance, offset = pssm.dist_pearson(pssm_reb1)
>>> print("distance = %5.3g" % distance)
distance = 0.239
>>> print(offset)
-2
```

This means that the best PCC between motif `m` and `m_reb1` is obtained with the following alignment:

```
m:      bbTACGCbb
m_reb1: GTTACCCGG
```

where `b` stands for background distribution. The PCC itself is roughly $1 - 0.239 = 0.761$.

17.9 *De novo* motif finding

Currently, Biopython has only limited support for *de novo* motif finding. Namely, we support running `xxmotif` and also parsing of MEME. Since the number of motif finding tools is growing rapidly, contributions of new parsers are welcome.

17.9.1 MEME

Let's assume, you have run MEME on sequences of your choice with your favorite parameters and saved the output in the file `meme.out`. You can retrieve the motifs reported by MEME by running the following piece of code:

```
>>> from Bio import motifs
>>> with open("meme.psp_test.classic.zoops.xml") as handle:
...     motifsM = motifs.parse(handle, "meme")
...
```

```
>>> motifsM
[<Bio.motifs.meme.Motif object at 0xc356b0>]
```

Besides the most wanted list of motifs, the result object contains more useful information, accessible through properties with self-explanatory names:

- `.alphabet`
- `.datafile`
- `.sequences`
- `.version`
- `.command`

The motifs returned by the MEME Parser can be treated exactly like regular Motif objects (with instances), they also provide some extra functionality, by adding additional information about the instances.

```
>>> motifsM[0].consensus
Seq('GCTTATGTAA')
>>> motifsM[0].alignment.sequences[0].sequence_name
'iYFL005W'
>>> motifsM[0].alignment.sequences[0].sequence_id
'sequence_15'
>>> motifsM[0].alignment.sequences[0].start
480
>>> motifsM[0].alignment.sequences[0].strand
'+'

>>> motifsM[0].alignment.sequences[0].pvalue
1.97e-06
```

17.10 Useful links

- Sequence motif in wikipedia
- PWM in wikipedia
- Consensus sequence in wikipedia
- Comparison of different motif finding programs

Chapter 18

Cluster analysis

Cluster analysis is the grouping of items into clusters based on the similarity of the items to each other. In bioinformatics, clustering is widely used in gene expression data analysis to find groups of genes with similar gene expression profiles. This may identify functionally related genes, as well as suggest the function of presently unknown genes.

The Biopython module `Bio.Cluster` provides commonly used clustering algorithms and was designed with the application to gene expression data in mind. However, this module can also be used for cluster analysis of other types of data. `Bio.Cluster` and the underlying C Clustering Library is described by De Hoon *et al.* [10].

The following four clustering approaches are implemented in `Bio.Cluster`:

- Hierarchical clustering (pairwise centroid-, single-, complete-, and average-linkage);
- k-means, k-medians, and k-medoids clustering;
- Self-Organizing Maps;
- Principal Component Analysis.

Data representation

The data to be clustered are represented by a $n \times m$ Numerical Python array `data`. Within the context of gene expression data clustering, typically the rows correspond to different genes whereas the columns correspond to different experimental conditions. The clustering algorithms in `Bio.Cluster` can be applied both to rows (genes) and to columns (experiments).

Missing values

The $n \times m$ Numerical Python integer array `mask` indicates if any of the values in `data` are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing and is ignored in the analysis.

Random number generator

The k-means/medians/medoids clustering algorithms and Self-Organizing Maps (SOMs) include the use of a random number generator. The uniform random number generator in `Bio.Cluster` is based on the algorithm by L'Ecuyer [27], while random numbers following the binomial distribution are generated using the BTPE algorithm by Kachitvichyanukul and Schmeiser [23]. The random number generator is initialized automatically during its first call. As this random number generator uses a combination of two multiplicative linear congruential generators, two (integer) seeds are needed for initialization, for which we use the system-supplied random number generator `rand` (in the C standard library). We initialize this generator by calling

`srand` with the epoch time in seconds, and use the first two random numbers generated by `rand` as seeds for the uniform random number generator in `Bio.Cluster`.

18.1 Distance functions

In order to cluster items into groups based on their similarity, we should first define what exactly we mean by *similar*. `Bio.Cluster` provides eight distance functions, indicated by a single character, to measure similarity, or conversely, distance:

- `'e'`: Euclidean distance;
- `'b'`: City-block distance.
- `'c'`: Pearson correlation coefficient;
- `'a'`: Absolute value of the Pearson correlation coefficient;
- `'u'`: Uncentered Pearson correlation (equivalent to the cosine of the angle between two data vectors);
- `'x'`: Absolute uncentered Pearson correlation;
- `'s'`: Spearman's rank correlation;
- `'k'`: Kendall's τ.

The first two are true distance functions that satisfy the triangle inequality:

$$d(\underline{u}, \underline{v}) \leq d(\underline{u}, \underline{w}) + d(\underline{w}, \underline{v}) \text{ for all } \underline{u}, \underline{v}, \underline{w},$$

and are therefore referred to as *metrics*. In everyday language, this means that the shortest distance between two points is a straight line.

The remaining six distance measures are related to the correlation coefficient, where the distance d is defined in terms of the correlation r by $d = 1 - r$. Note that these distance functions are *semi-metrics* that do not satisfy the triangle inequality. For example, for

$$\underline{u} = (1, 0, -1);$$

$$\underline{v} = (1, 1, 0);$$

$$\underline{w} = (0, 1, 1);$$

we find a Pearson distance $d(\underline{u}, \underline{w}) = 1.8660$, while $d(\underline{u}, \underline{v}) + d(\underline{v}, \underline{w}) = 1.6340$.

Euclidean distance

In `Bio.Cluster`, we define the Euclidean distance as

$$d = \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2.$$

Only those terms are included in the summation for which both x_i and y_i are present, and the denominator n is chosen accordingly. As the expression data x_i and y_i are subtracted directly from each other, we should make sure that the expression data are properly normalized when using the Euclidean distance.

City-block distance

The city-block distance, alternatively known as the Manhattan distance, is related to the Euclidean distance. Whereas the Euclidean distance corresponds to the length of the shortest path between two points, the city-block distance is the sum of distances along each dimension. As gene expression data tend to have missing values, in `Bio.Cluster` we define the city-block distance as the sum of distances divided by the number of dimensions:

$$d = \frac{1}{n}\sum_{i=1}^{n} |x_i - y_i| .$$

This is equal to the distance you would have to walk between two points in a city, where you have to walk along city blocks. As for the Euclidean distance, the expression data are subtracted directly from each other, and we should therefore make sure that they are properly normalized.

The Pearson correlation coefficient

The Pearson correlation coefficient is defined as

$$r = \frac{1}{n}\sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{\sigma_x}\right)\left(\frac{y_i - \bar{y}}{\sigma_y}\right),$$

in which $\bar{x}, \bar{y}$ are the sample mean of x and y respectively, and σ_x, σ_y are the sample standard deviation of x and y. The Pearson correlation coefficient is a measure for how well a straight line can be fitted to a scatterplot of x and y. If all the points in the scatterplot lie on a straight line, the Pearson correlation coefficient is either +1 or -1, depending on whether the slope of line is positive or negative. If the Pearson correlation coefficient is equal to zero, there is no correlation between x and y.

The *Pearson distance* is then defined as

$$d_{\mathrm{P}} \equiv 1 - r.$$

As the Pearson correlation coefficient lies between -1 and 1, the Pearson distance lies between 0 and 2.

Absolute Pearson correlation

By taking the absolute value of the Pearson correlation, we find a number between 0 and 1. If the absolute value is 1, all the points in the scatter plot lie on a straight line with either a positive or a negative slope. If the absolute value is equal to zero, there is no correlation between x and y.

The corresponding distance is defined as

$$d_{\mathrm{A}} \equiv 1 - |r| ,$$

where r is the Pearson correlation coefficient. As the absolute value of the Pearson correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

In the context of gene expression experiments, the absolute correlation is equal to 1 if the gene expression profiles of two genes are either exactly the same or exactly opposite. The absolute correlation coefficient should therefore be used with care.

Uncentered correlation (cosine of the angle)

In some cases, it may be preferable to use the *uncentered correlation* instead of the regular Pearson correlation coefficient. The uncentered correlation is defined as

$$r_{\mathrm{U}} = \frac{1}{n}\sum_{i=1}^{n} \left(\frac{x_i}{\sigma_x^{(0)}}\right)\left(\frac{y_i}{\sigma_y^{(0)}}\right),$$

where

$$\sigma_x^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2};$$
$$\sigma_y^{(0)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2}.$$

This is the same expression as for the regular Pearson correlation coefficient, except that the sample means $\bar{x}, \bar{y}$ are set equal to zero. The uncentered correlation may be appropriate if there is a zero reference state. For instance, in the case of gene expression data given in terms of log-ratios, a log-ratio equal to zero corresponds to the green and red signal being equal, which means that the experimental manipulation did not affect the gene expression.

The distance corresponding to the uncentered correlation coefficient is defined as

$$d_{\mathrm{U}} \equiv 1 - r_{\mathrm{U}},$$

where r_{U} is the uncentered correlation. As the uncentered correlation coefficient lies between -1 and 1, the corresponding distance lies between 0 and 2.

The uncentered correlation is equal to the cosine of the angle of the two data vectors in n-dimensional space, and is often referred to as such.

Absolute uncentered correlation

As for the regular Pearson correlation, we can define a distance measure using the absolute value of the uncentered correlation:

$$d_{\mathrm{AU}} \equiv 1 - \left|r_{\mathrm{U}}\right|,$$

where r_{U} is the uncentered correlation coefficient. As the absolute value of the uncentered correlation coefficient lies between 0 and 1, the corresponding distance lies between 0 and 1 as well.

Geometrically, the absolute value of the uncentered correlation is equal to the cosine between the supporting lines of the two data vectors (i.e., the angle without taking the direction of the vectors into consideration).

Spearman rank correlation

The Spearman rank correlation is an example of a non-parametric similarity measure, and tends to be more robust against outliers than the Pearson correlation.

To calculate the Spearman rank correlation, we replace each data value by their rank if we would order the data in each vector by their value. We then calculate the Pearson correlation between the two rank vectors instead of the data vectors.

As in the case of the Pearson correlation, we can define a distance measure corresponding to the Spearman rank correlation as

$$d_{\mathrm{S}} \equiv 1 - r_{\mathrm{S}},$$

where r_{S} is the Spearman rank correlation.

Kendall's τ

Kendall's τ is another example of a non-parametric similarity measure. It is similar to the Spearman rank correlation, but instead of the ranks themselves only the relative ranks are used to calculate τ (see Snedecor & Cochran [43]).

We can define a distance measure corresponding to Kendall's τ as

$$d_{\mathrm{K}} \equiv 1 - \tau.$$

As Kendall's τ is always between -1 and 1, the corresponding distance will be between 0 and 2.

Weighting

For most of the distance functions available in `Bio.Cluster`, a weight vector can be applied. The weight vector contains weights for the items in the data vector. If the weight for item i is w_i, then that item is treated as if it occurred w_i times in the data. The weight do not have to be integers.

Calculating the distance matrix

The distance matrix is a square matrix with all pairwise distances between the items in `data`, and can be calculated by the function `distancematrix` in the `Bio.Cluster` module:

```
>>> from Bio.Cluster import distancematrix
>>> matrix = distancematrix(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if the distances between the rows of `data` are to be calculated (`transpose` is `False`), or between the columns of `data` (`transpose` is `True`).
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

To save memory, the distance matrix is returned as a list of 1D arrays. The number of columns in each row is equal to the row number. Hence, the first row has zero elements. For example,

```
>>> from numpy import array
>>> from Bio.Cluster import distancematrix
>>> data = array([[0, 1,  2,  3],
...               [4, 5,  6,  7],
...               [8, 9, 10, 11],
...               [1, 2,  3,  4]])  # fmt: skip
...
>>> distances = distancematrix(data, dist="e")
```

yields a distance matrix

```
>>> distances
[array([], dtype=float64), array([ 16.]), array([ 64.,  16.]), array([  1.,   9.,  49.])]
```

which can be rewritten as

```
[array([], dtype=float64), array([16.]), array([64.0, 16.0]), array([1.0, 9.0, 49.0])]
```

This corresponds to the distance matrix:

$$\begin{pmatrix} 0 & 16 & 64 & 1 \\ 16 & 0 & 16 & 9 \\ 64 & 16 & 0 & 49 \\ 1 & 9 & 49 & 0 \end{pmatrix}.$$

18.2 Calculating cluster properties

Calculating the cluster centroids

The centroid of a cluster can be defined either as the mean or as the median of each dimension over all cluster items. The function `clustercentroids` in `Bio.Cluster` can be used to calculate either:

```
>>> from Bio.Cluster import clustercentroids
>>> cdata, cmask = clustercentroids(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `clusterid` (default: `None`)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is `None`, then all items are assumed to belong to the same cluster.
- `method` (default: `'a'`)
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.
- `transpose` (default: `0`)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `(cdata, cmask)`. The centroid data are stored in the 2D Numerical Python array `cdata`, with missing data indicated by the 2D Numerical Python integer array `cmask`. The dimensions of these arrays are (number of clusters, number of columns) if `transpose` is `0`, or (number of rows, number of clusters) if `transpose` is `1`. Each row (if `transpose` is `0`) or column (if `transpose` is `1`) contains the averaged data corresponding to the centroid of each cluster.

Calculating the distance between clusters

Given a distance function between *items*, we can define the distance between two *clusters* in several ways. The distance between the arithmetic means of the two clusters is used in pairwise centroid-linkage clustering and in k-means clustering. In k-medoids clustering, the distance between the medians of the two clusters is used instead. The shortest pairwise distance between items of the two clusters is used in pairwise single-linkage clustering, while the longest pairwise distance is used in pairwise maximum-linkage clustering. In pairwise average-linkage clustering, the distance between two clusters is defined as the average over the pairwise distances.

To calculate the distance between two clusters, use

```
>>> from Bio.Cluster import clusterdistance
>>> distance = clusterdistance(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.

- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `index1` (default: `0`)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.
- `index2` (default: `0`)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one items i can be represented either as a list `[i]`, or as an integer `i`.
- `method` (default: `'a'`)
  Specifies how the distance between clusters is defined:
  - `'a'`: Distance between the two cluster centroids (arithmetic mean);
  - `'m'`: Distance between the two cluster centroids (median);
  - `'s'`: Shortest pairwise distance between items in the two clusters;
  - `'x'`: Longest pairwise distance between items in the two clusters;
  - `'v'`: Average over the pairwise distances between items in the two clusters.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).
- `transpose` (default: `0`)
  If `transpose` is `False`, calculate the distance between the rows of `data`. If `transpose` is `True`, calculate the distance between the columns of `data`.

18.3 Partitioning algorithms

Partitioning algorithms divide items into k clusters such that the sum of distances over the items to their cluster centers is minimal. The number of clusters k is specified by the user. Three partitioning algorithms are available in `Bio.Cluster`:

- k-means clustering
- k-medians clustering
- k-medoids clustering

These algorithms differ in how the cluster center is defined. In k-means clustering, the cluster center is defined as the mean data vector averaged over all items in the cluster. Instead of the mean, in k-medians clustering the median is calculated for each dimension in the data vector. Finally, in k-medoids clustering the cluster center is defined as the item which has the smallest sum of distances to the other items in the cluster. This clustering algorithm is suitable for cases in which the distance matrix is known but the original data matrix is not available, for example when clustering proteins based on their structural similarity.

The expectation-maximization (EM) algorithm is used to find this partitioning into k groups. In the initialization of the EM algorithm, we randomly assign items to clusters. To ensure that no empty clusters are produced, we use the binomial distribution to randomly choose the number of items in each cluster to be one or more. We then randomly permute the cluster assignments to items such that each item has an equal probability to be in any cluster. Each cluster is thus guaranteed to contain at least one item.

We then iterate:

- Calculate the centroid of each cluster, defined as either the mean, the median, or the medoid of the cluster;
- Calculate the distances of each item to the cluster centers;
- For each item, determine which cluster centroid is closest;
- Reassign each item to its closest cluster, or stop the iteration if no further item reassignments take place.

To avoid clusters becoming empty during the iteration, in k-means and k-medians clustering the algorithm keeps track of the number of items in each cluster, and prohibits the last remaining item in a cluster from being reassigned to a different cluster. For k-medoids clustering, such a check is not needed, as the item that functions as the cluster centroid has a zero distance to itself, and will therefore never be closer to a different cluster.

As the initial assignment of items to clusters is done randomly, usually a different clustering solution is found each time the EM algorithm is executed. To find the optimal clustering solution, the k-means algorithm is repeated many times, each time starting from a different initial random clustering. The sum of distances of the items to their cluster center is saved for each run, and the solution with the smallest value of this sum will be returned as the overall clustering solution.

How often the EM algorithm should be run depends on the number of items being clustered. As a rule of thumb, we can consider how often the optimal solution was found; this number is returned by the partitioning algorithms as implemented in this library. If the optimal solution was found many times, it is unlikely that better solutions exist than the one that was found. However, if the optimal solution was found only once, there may well be other solutions with a smaller within-cluster sum of distances. If the number of items is large (more than several hundreds), it may be difficult to find the globally optimal solution.

The EM algorithm terminates when no further reassignments take place. We noticed that for some sets of initial cluster assignments, the EM algorithm fails to converge due to the same clustering solution reappearing periodically after a small number of iteration steps. We therefore check for the occurrence of such periodic solutions during the iteration. After a given number of iteration steps, the current clustering result is saved as a reference. By comparing the clustering result after each subsequent iteration step to the reference state, we can determine if a previously encountered clustering result is found. In such a case, the iteration is halted. If after a given number of iterations the reference state has not yet been encountered, the current clustering solution is saved to be used as the new reference state. Initially, ten iteration steps are executed before resaving the reference state. This number of iteration steps is doubled each time, to ensure that periodic behavior with longer periods can also be detected.

k-means and k-medians

The k-means and k-medians algorithms are implemented as the function `kcluster` in `Bio.Cluster`:

```
>>> from Bio.Cluster import kcluster
>>> clusterid, error, nfound = kcluster(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `nclusters` (default: `2`)
  The number of clusters k.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.

- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.

- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.

- `npass` (default: `1`)
  The number of times the k-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.

- `method` (default: `a`)
  describes how the center of a cluster is found:

  - `method=='a'`: arithmetic mean (k-means clustering);
  - `method=='m'`: median (k-medians clustering).

  For other values of `method`, the arithmetic mean is used.

- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1). Whereas all eight distance measures are accepted by `kcluster`, from a theoretical viewpoint it is best to use the Euclidean distance for the k-means algorithm, and the city-block distance for k-medians.

- `initialid` (default: `None`)
  Specifies the initial clustering to be used for the EM algorithm. If `initialid` is `None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an integer array containing the number of the cluster to which each row or cluster was assigned, `error` is the within-cluster sum of distances for the optimal clustering solution, and `nfound` is the number of times this optimal solution was found.

k-medoids clustering

The `kmedoids` routine performs k-medoids clustering on a given set of items, using the distance matrix and the number of clusters passed by the user:

```
>>> from Bio.Cluster import kmedoids
>>> clusterid, error, nfound = kmedoids(distance)
```

where the following arguments are defined: , nclusters=2, npass=1, initialid=None)—

- `distance` (required)
  The matrix containing the distances between the items; this matrix can be specified in three ways:

  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):

    ```
    distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])
    ```

  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:

    ```
    distance = array([1.1, 2.3, 4.5])
    ```

– as a list containing the rows of the left-lower part of the distance matrix:

```
distance = [array([]), array([1.1]), array([2.3, 4.5])]
```

These three expressions correspond to the same distance matrix.

- `nclusters` (default: `2`)
  The number of clusters k.

- `npass` (default: `1`)
  The number of times the k-medoids clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored, as the clustering algorithm behaves deterministically in that case.

- `initialid` (default: `None`)
  Specifies the initial clustering to be used for the EM algorithm. If `initialid` is `None`, then a different random initial clustering is used for each of the `npass` runs of the EM algorithm. If `initialid` is not `None`, then it should be equal to a 1D array containing the cluster number (between `0` and `nclusters-1`) for each item. Each cluster should contain at least one item. With the initial clustering specified, the EM algorithm is deterministic.

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an array containing the number of the cluster to which each item was assigned, `error` is the within-cluster sum of distances for the optimal k-medoids clustering solution, and `nfound` is the number of times the optimal solution was found. Note that the cluster number in `clusterid` is defined as the item number of the item representing the cluster centroid.

18.4 Hierarchical clustering

Hierarchical clustering methods are inherently different from the k-means clustering method. In hierarchical clustering, the similarity in the expression profile between genes or experimental conditions are represented in the form of a tree structure. This tree structure can be shown graphically by programs such as Treeview and Java Treeview, which has contributed to the popularity of hierarchical clustering in the analysis of gene expression data.

The first step in hierarchical clustering is to calculate the distance matrix, specifying all the distances between the items to be clustered. Next, we create a node by joining the two closest items. Subsequent nodes are created by pairwise joining of items or nodes based on the distance between them, until all items belong to the same node. A tree structure can then be created by retracing which items and nodes were merged. Unlike the EM algorithm, which is used in k-means clustering, the complete process of hierarchical clustering is deterministic.

Several flavors of hierarchical clustering exist, which differ in how the distance between subnodes is defined in terms of their members. In `Bio.Cluster`, pairwise single, maximum, average, and centroid linkage are available.

- In pairwise single-linkage clustering, the distance between two nodes is defined as the shortest distance among the pairwise distances between the members of the two nodes.

- In pairwise maximum-linkage clustering, alternatively known as pairwise complete-linkage clustering, the distance between two nodes is defined as the longest distance among the pairwise distances between the members of the two nodes.

- In pairwise average-linkage clustering, the distance between two nodes is defined as the average over all pairwise distances between the items of the two nodes.

- In pairwise centroid-linkage clustering, the distance between two nodes is defined as the distance between their centroids. The centroids are calculated by taking the mean over all the items in a cluster. As the distance from each newly formed node to existing nodes and items need to be calculated at each step, the computing time of pairwise centroid-linkage clustering may be significantly longer than for the other hierarchical clustering methods. Another peculiarity is that (for a distance measure based on the Pearson correlation), the distances do not necessarily increase when going up in the clustering tree, and may even decrease. This is caused by an inconsistency between the centroid calculation and the distance calculation when using the Pearson correlation: Whereas the Pearson correlation effectively normalizes the data for the distance calculation, no such normalization occurs for the centroid calculation.

For pairwise single-, complete-, and average-linkage clustering, the distance between two nodes can be found directly from the distances between the individual items. Therefore, the clustering algorithm does not need access to the original gene expression data, once the distance matrix is known. For pairwise centroid-linkage clustering, however, the centroids of newly formed subnodes can only be calculated from the original data and not from the distance matrix.

The implementation of pairwise single-linkage hierarchical clustering is based on the SLINK algorithm [41], which is much faster and more memory-efficient than a straightforward implementation of pairwise single-linkage clustering. The clustering result produced by this algorithm is identical to the clustering solution found by the conventional single-linkage algorithm. The single-linkage hierarchical clustering algorithm implemented in this library can be used to cluster large gene expression data sets, for which conventional hierarchical clustering algorithms fail due to excessive memory requirements and running time.

Representing a hierarchical clustering solution

The result of hierarchical clustering consists of a tree of nodes, in which each node joins two items or subnodes. Usually, we are not only interested in which items or subnodes are joined at each node, but also in their similarity (or distance) as they are joined. To store one node in the hierarchical clustering tree, we make use of the class `Node`, which defined in `Bio.Cluster`. An instance of `Node` has three attributes:

- `left`
- `right`
- `distance`

Here, `left` and `right` are integers referring to the two items or subnodes that are joined at this node, and `distance` is the distance between them. The items being clustered are numbered from 0 to (number of items $-$ 1), while clusters are numbered from -1 to $-$ (number of items $-$ 1). Note that the number of nodes is one less than the number of items.

To create a new `Node` object, we need to specify `left` and `right`; `distance` is optional.

```
>>> from Bio.Cluster import Node
>>> Node(2, 3)
(2, 3): 0
>>> Node(2, 3, 0.91)
(2, 3): 0.91
```

The attributes `left`, `right`, and `distance` of an existing `Node` object can be modified directly:

```
>>> node = Node(4, 5)
>>> node.left = 6
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```

An error is raised if `left` and `right` are not integers, or if `distance` cannot be converted to a floating-point value.

The Python class `Tree` represents a full hierarchical clustering solution. A `Tree` object can be created from a list of `Node` objects:

```
>>> from Bio.Cluster import Node, Tree
>>> nodes = [Node(1, 2, 0.2), Node(0, 3, 0.5), Node(-2, 4, 0.6), Node(-1, -3, 0.9)]
>>> tree = Tree(nodes)
>>> print(tree)
(1, 2): 0.2
(0, 3): 0.5
(-2, 4): 0.6
(-1, -3): 0.9
```

The `Tree` initializer checks if the list of nodes is a valid hierarchical clustering result:

```
>>> nodes = [Node(1, 2, 0.2), Node(0, 2, 0.5)]
>>> Tree(nodes)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: Inconsistent tree
```

Individual nodes in a `Tree` object can be accessed using square brackets:

```
>>> nodes = [Node(1, 2, 0.2), Node(0, -1, 0.5)]
>>> tree = Tree(nodes)
>>> tree[0]
(1, 2): 0.2
>>> tree[1]
(0, -1): 0.5
>>> tree[-1]
(0, -1): 0.5
```

As a `Tree` object is immutable, we cannot change individual nodes in a `Tree` object. However, we can convert the tree to a list of nodes, modify this list, and create a new tree from this list:

```
>>> tree = Tree([Node(1, 2, 0.1), Node(0, -1, 0.5), Node(-2, 3, 0.9)])
>>> print(tree)
(1, 2): 0.1
(0, -1): 0.5
(-2, 3): 0.9
>>> nodes = tree[:]
>>> nodes[0] = Node(0, 1, 0.2)
>>> nodes[1].left = 2
>>> tree = Tree(nodes)
>>> print(tree)
(0, 1): 0.2
(2, -1): 0.5
(-2, 3): 0.9
```

This guarantees that any `Tree` object is always well-formed.

To display a hierarchical clustering solution with visualization programs such as Java Treeview, it is better to scale all node distances such that they are between zero and one. This can be accomplished by calling the `scale` method on an existing `Tree` object:

```
>>> tree.scale()
```

This method takes no arguments, and returns `None`.

Before drawing the tree, you may also want to reorder the tree nodes. A hierarchical clustering solution of n items can be drawn as 2^{n-1} different but equivalent dendrograms by switching the left and right subnode at each node. The `tree.sort(order)` method visits each node in the hierarchical clustering tree and verifies if the average order value of the left subnode is less than or equal to the average order value of the right subnode. If not, the left and right subnodes are exchanged. Here, the order values of the items are given by the user. In the resulting dendrogram, items in the left-to-right order will tend to have increasing order values. The method will return the indices of the elements in the left-to-right order after sorting:

```
>>> indices = tree.sort(order)
```

such that item `indices[i]` will occur at position i in the dendrogram.

After hierarchical clustering, the items can be grouped into k clusters based on the tree structure stored in the `Tree` object by cutting the tree:

```
>>> clusterid = tree.cut(nclusters=1)
```

where `nclusters` (defaulting to `1`) is the desired number of clusters k. This method ignores the top $k-1$ linking events in the tree structure, resulting in k separated clusters of items. The number of clusters k should be positive, and less than or equal to the number of items. This method returns an array `clusterid` containing the number of the cluster to which each item is assigned. Clusters are numbered 0 to $k-1$ in their left-to-right order in the dendrogram.

Performing hierarchical clustering

To perform hierarchical clustering, use the `treecluster` function in `Bio.Cluster`.

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(data)
```

where the following arguments are defined:

- `data`
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  The weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `False`) or columns (`transpose` is `True`) are to be clustered.
- `method` (default: `'m'`)
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering

- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

To apply hierarchical clustering on a precalculated distance matrix, specify the `distancematrix` argument when calling `treecluster` function instead of the `data` argument:

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(distancematrix=distance)
```

In this case, the following arguments are defined:

- `distancematrix`
  The distance matrix, which can be specified in three ways:
  - as a 2D Numerical Python array (in which only the left-lower part of the array will be accessed):
    `distance = array([[0.0, 1.1, 2.3], [1.1, 0.0, 4.5], [2.3, 4.5, 0.0]])`
  - as a 1D Numerical Python array containing consecutively the distances in the left-lower part of the distance matrix:
    `distance = array([1.1, 2.3, 4.5])`
  - as a list containing the rows of the left-lower part of the distance matrix:
    `distance = [array([]), array([1.1]), array([2.3, 4.5])]`

  These three expressions correspond to the same distance matrix. As `treecluster` may shuffle the values in the distance matrix as part of the clustering algorithm, be sure to save this array in a different variable before calling `treecluster` if you need it later.

- `method`
  The linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='a'`: pairwise average-linkage clustering

  While pairwise single-, maximum-, and average-linkage clustering can be calculated from the distance matrix alone, pairwise centroid-linkage cannot.

When calling `treecluster`, either `data` or `distancematrix` should be `None`.

This function returns a `Tree` object. This object contains (number of items − 1) nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one item or subnode, and `distance` the distance between them. Items are numbered from 0 to (number of items − 1), while clusters are numbered -1 to − (number of items − 1).

18.5 Self-Organizing Maps

Self-Organizing Maps (SOMs) were invented by Kohonen to describe neural networks (see for instance Kohonen, 1997 [25]). Tamayo (1999) first applied Self-Organizing Maps to gene expression data [46].

SOMs organize items into clusters that are situated in some topology. Usually a rectangular topology is chosen. The clusters generated by SOMs are such that neighboring clusters in the topology are more similar to each other than clusters far from each other in the topology.

The first step to calculate a SOM is to randomly assign a data vector to each cluster in the topology. If rows are being clustered, then the number of elements in each data vector is equal to the number of columns.

An SOM is then generated by taking rows one at a time, and finding which cluster in the topology has the closest data vector. The data vector of that cluster, as well as those of the neighboring clusters, are adjusted using the data vector of the row under consideration. The adjustment is given by

$$\Delta \underline{x}_{\text{cell}} = \tau \cdot \left(\underline{x}_{\text{row}} - \underline{x}_{\text{cell}} \right).$$

The parameter τ is a parameter that decreases at each iteration step. We have used a simple linear function of the iteration step:

$$\tau = \tau_{\text{init}} \cdot \left(1 - \frac{i}{n} \right),$$

τ_{init} is the initial value of τ as specified by the user, i is the number of the current iteration step, and n is the total number of iteration steps to be performed. While changes are made rapidly in the beginning of the iteration, at the end of iteration only small changes are made.

All clusters within a radius R are adjusted to the gene under consideration. This radius decreases as the calculation progresses as

$$R = R_{\text{max}} \cdot \left(1 - \frac{i}{n} \right),$$

in which the maximum radius is defined as

$$R_{\text{max}} = \sqrt{N_x^2 + N_y^2},$$

where (N_x, N_y) are the dimensions of the rectangle defining the topology.

The function `somcluster` implements the complete algorithm to calculate a Self-Organizing Map on a rectangular grid. First it initializes the random number generator. The node data are then initialized using the random number generator. The order in which genes or samples are used to modify the SOM is also randomized. The total number of iterations in the SOM algorithm is specified by the user.

To run `somcluster`, use

```
>>> from Bio.Cluster import somcluster
>>> clusterid, celldata = somcluster(data)
```

where the following arguments are defined:

- `data` (required)
  Array containing the data for the items.
- `mask` (default: `None`)
  Array of integers showing which data are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If `mask` is `None`, then all data are present.
- `weight` (default: `None`)
  contains the weights to be used when calculating distances. If `weight` is `None`, then equal weights are assumed.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `nxgrid, nygrid` (default: `2, 1`)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.
- `inittau` (default: `0.02`)
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.

- `niter` (default: `1`)
  The number of iterations to be performed.

- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns the tuple `(clusterid, celldata)`:

- `clusterid`:
  An array with two columns, where the number of rows is equal to the number of items that were clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the item was assigned.

- `celldata`:
  An array with dimensions (`nxgrid`, `nygrid`, number of columns) if rows are being clustered, or (`nxgrid`, `nygrid`, number
  if columns are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates `[ix][iy]`.

18.6 Principal Component Analysis

Principal Component Analysis (PCA) is a widely used technique for analyzing multivariate data. A practical example of applying Principal Component Analysis to gene expression data is presented by Yeung and Ruzzo (2001) [52].

In essence, PCA is a coordinate transformation in which each row in the data matrix is written as a linear sum over basis vectors called principal components, which are ordered and chosen such that each maximally explains the remaining variance in the data vectors. For example, an $n \times 3$ data matrix can be represented as an ellipsoidal cloud of n points in three dimensional space. The first principal component is the longest axis of the ellipsoid, the second principal component the second longest axis of the ellipsoid, and the third principal component is the shortest axis. Each row in the data matrix can be reconstructed as a suitable linear combination of the principal components. However, in order to reduce the dimensionality of the data, usually only the most important principal components are retained. The remaining variance present in the data is then regarded as unexplained variance.

The principal components can be found by calculating the eigenvectors of the covariance matrix of the data. The corresponding eigenvalues determine how much of the variance present in the data is explained by each principal component.

Before applying principal component analysis, typically the mean is subtracted from each column in the data matrix. In the example above, this effectively centers the ellipsoidal cloud around its centroid in 3D space, with the principal components describing the variation of points in the ellipsoidal cloud with respect to their centroid.

The function `pca` below first uses the singular value decomposition to calculate the eigenvalues and eigenvectors of the data matrix. The singular value decomposition is implemented as a translation in C of the Algol procedure `svd` [14], which uses Householder bidiagonalization and a variant of the QR algorithm. The principal components, the coordinates of each data vector along the principal components, and the eigenvalues corresponding to the principal components are then evaluated and returned in decreasing order of the magnitude of the eigenvalue. If data centering is desired, the mean should be subtracted from each column in the data matrix before calling the `pca` routine.

To apply Principal Component Analysis to a rectangular matrix `data`, use

```
>>> from Bio.Cluster import pca
>>> columnmean, coordinates, components, eigenvalues = pca(data)
```

This function returns a tuple `columnmean, coordinates, components, eigenvalues`:

- `columnmean`
  Array containing the mean over each column in `data`.
- `coordinates`
  The coordinates of each row in `data` with respect to the principal components.
- `components`
  The principal components.
- `eigenvalues`
  The eigenvalues corresponding to each of the principal components.

The original matrix `data` can be recreated by calculating `columnmean + dot(coordinates, components)`.

18.7 Handling Cluster/TreeView-type files

Cluster/TreeView are GUI-based codes for clustering gene expression data. They were originally written by Michael Eisen while at Stanford University [12]. `Bio.Cluster` contains functions for reading and writing data files that correspond to the format specified for Cluster/TreeView. In particular, by saving a clustering result in that format, TreeView can be used to visualize the clustering results. We recommend using Alok Saldanha's http://jtreeview.sourceforge.net/Java TreeView program [38], which can display hierarchical as well as k-means clustering results.

An object of the class `Record` contains all information stored in a Cluster/TreeView-type data file. To store the information contained in the data file in a `Record` object, we first open the file and then read it:

```
>>> from Bio import Cluster
>>> with open("mydatafile.txt") as handle:
...     record = Cluster.read(handle)
...
```

This two-step process gives you some flexibility in the source of the data. For example, you can use

```
>>> import gzip  # Python standard library
>>> handle = gzip.open("mydatafile.txt.gz", "rt")
```

to open a gzipped file, or

```
>>> from urllib.request import urlopen
>>> from io import TextIOWrapper
>>> url = "https://raw.githubusercontent.com/biopython/biopython/master/Tests/Cluster/cyano.txt"
>>> handle = TextIOWrapper(urlopen(url))
```

to open a file stored on the Internet before calling `read`.

The `read` command reads the tab-delimited text file `mydatafile.txt` containing gene expression data in the format specified for Michael Eisen's Cluster/TreeView program. In this file format, rows represent genes and columns represent samples or observations. For a simple time course, a minimal input file would look like this:
Each row (gene) has an identifier that always goes in the first column. In this example, we are using yeast open reading frame codes. Each column (sample) has a label in the first row. In this example, the labels describe the time at which a sample was taken. The first column of the first row contains a special field that tells the program what kind of objects are in each row. In this case, YORF stands for yeast open reading frame. This field can be any alphanumeric value. The remaining cells in the table contain data for the appropriate gene and sample. The 5.8 in row 2 column 4 means that the observed value for gene YAL001C

YORF	0 minutes	30 minutes	1 hour	2 hours	4 hours
YAL001C	1	1.3	2.4	5.8	2.4
YAL002W	0.9	0.8	0.7	0.5	0.2
YAL003W	0.8	2.1	4.2	10.1	10.1
YAL005C	1.1	1.3	0.8		0.4
YAL010C	1.2	1	1.1	4.5	8.3

YORF	NAME	GWEIGHT	GORDER	0	30	1	2	4
EWEIGHT				1	1	1	1	0
EORDER				5	3	2	1	1
YAL001C	TFIIIC 138 KD SUBUNIT	1	1	1	1.3	2.4	5.8	2.4
YAL002W	UNKNOWN	0.4	3	0.9	0.8	0.7	0.5	0.2
YAL003W	ELONGATION FACTOR EF1-BETA	0.4	2	0.8	2.1	4.2	10.1	10.1
YAL005C	CYTOSOLIC HSP70	0.4	5	1.1	1.3	0.8		0.4

at 2 hours was 5.8. Missing values are acceptable and are designated by empty cells (e.g. YAL004C at 2 hours).

The input file may contain additional information. A maximal input file would look like this:
The added columns NAME, GWEIGHT, and GORDER and rows EWEIGHT and EORDER are optional. The NAME column allows you to specify a label for each gene that is distinct from the ID in column 1.

A `Record` object has the following attributes:

- `data`
  The data array containing the gene expression data. Genes are stored row-wise, while samples are stored column-wise.

- `mask`
  This array shows which elements in the `data` array, if any, are missing. If `mask[i, j] == 0`, then `data[i, j]` is missing. If no data were found to be missing, `mask` is set to `None`.

- `geneid`
  This is a list containing a unique description for each gene (i.e., ORF numbers).

- `genename`
  This is a list containing a description for each gene (i.e., gene name). If not present in the data file, `genename` is set to `None`.

- `gweight`
  The weights that are to be used to calculate the distance in expression profile between genes. If not present in the data file, `gweight` is set to `None`.

- `gorder`
  The preferred order in which genes should be stored in an output file. If not present in the data file, `gorder` is set to `None`.

- `expid`
  This is a list containing a description of each sample, e.g. experimental condition.

- `eweight`
  The weights that are to be used to calculate the distance in expression profile between samples. If not present in the data file, `eweight` is set to `None`.

- `eorder`
  The preferred order in which samples should be stored in an output file. If not present in the data file, `eorder` is set to `None`.

- `uniqid`
  The string that was used instead of UNIQID in the data file.

After loading a `Record` object, each of these attributes can be accessed and modified directly. For example, the data can be log-transformed by taking the logarithm of `record.data`.

Calculating the distance matrix

To calculate the distance matrix between the items stored in the record, use

```
>>> matrix = record.distancematrix()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if the distances between the rows of `data` are to be calculated (`transpose` is `False`), or between the columns of `data` (`transpose` is `True`).

- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns the distance matrix as a list of rows, where the number of columns of each row is equal to the row number (see section 18.1).

Calculating the cluster centroids

To calculate the centroids of clusters of items stored in the record, use

```
>>> cdata, cmask = record.clustercentroids()
```

- `clusterid` (default: `None`)
  Vector of integers showing to which cluster each item belongs. If `clusterid` is not given, then all items are assumed to belong to the same cluster.

- `method` (default: `'a'`)
  Specifies whether the arithmetic mean (`method=='a'`) or the median (`method=='m'`) is used to calculate the cluster center.

- `transpose` (default: `0`)
  Determines if the centroids of the rows of `data` are to be calculated (`transpose` is `False`), or the centroids of the columns of `data` (`transpose` is `True`).

This function returns the tuple `cdata, cmask`; see section 18.2 for a description.

Calculating the distance between clusters

To calculate the distance between clusters of items stored in the record, use

```
>>> distance = record.clusterdistance()
```

where the following arguments are defined:

- `index1` (default: `0`)
  A list containing the indices of the items belonging to the first cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.
- `index2` (default: `0`)
  A list containing the indices of the items belonging to the second cluster. A cluster containing only one item i can be represented either as a list `[i]`, or as an integer `i`.
- `method` (default: `'a'`)
  Specifies how the distance between clusters is defined:
  - `'a'`: Distance between the two cluster centroids (arithmetic mean);
  - `'m'`: Distance between the two cluster centroids (median);
  - `'s'`: Shortest pairwise distance between items in the two clusters;
  - `'x'`: Longest pairwise distance between items in the two clusters;
  - `'v'`: Average over the pairwise distances between items in the two clusters.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).
- `transpose` (default: `0`)
  If `transpose` is `False`, calculate the distance between the rows of `data`. If `transpose` is `True`, calculate the distance between the columns of `data`.

Performing hierarchical clustering

To perform hierarchical clustering on the items stored in the record, use

```
>>> tree = record.treecluster()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if rows (`transpose` is `False`) or columns (`transpose` is `True`) are to be clustered.
- `method` (default: `'m'`)
  defines the linkage method to be used:
  - `method=='s'`: pairwise single-linkage clustering
  - `method=='m'`: pairwise maximum- (or complete-) linkage clustering
  - `method=='c'`: pairwise centroid-linkage clustering
  - `method=='a'`: pairwise average-linkage clustering
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).
- `transpose`
  Determines if genes or samples are being clustered. If `transpose` is `False`, genes (rows) are being clustered. If `transpose` is `True`, samples (columns) are clustered.

This function returns a `Tree` object. This object contains (number of items $-$ 1) nodes, where the number of items is the number of rows if rows were clustered, or the number of columns if columns were clustered. Each node describes a pairwise linking event, where the node attributes `left` and `right` each contain the number of one item or subnode, and `distance` the distance between them. Items are numbered from 0 to (number of items $-$ 1), while clusters are numbered -1 to $-$ (number of items $-$ 1).

Performing k-means or k-medians clustering

To perform k-means or k-medians clustering on the items stored in the record, use

```
>>> clusterid, error, nfound = record.kcluster()
```

where the following arguments are defined:

- `nclusters` (default: `2`)
  The number of clusters k.
- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `npass` (default: `1`)
  The number of times the k-means/-medians clustering algorithm is performed, each time with a different (random) initial condition. If `initialid` is given, the value of `npass` is ignored and the clustering algorithm is run only once, as it behaves deterministically in that case.
- `method` (default: `a`)
  describes how the center of a cluster is found:
  - `method=='a'`: arithmetic mean (k-means clustering);
  - `method=='m'`: median (k-medians clustering).

  For other values of `method`, the arithmetic mean is used.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns a tuple `(clusterid, error, nfound)`, where `clusterid` is an integer array containing the number of the cluster to which each row or cluster was assigned, `error` is the within-cluster sum of distances for the optimal clustering solution, and `nfound` is the number of times this optimal solution was found.

Calculating a Self-Organizing Map

To calculate a Self-Organizing Map of the items stored in the record, use

```
>>> clusterid, celldata = record.somcluster()
```

where the following arguments are defined:

- `transpose` (default: `0`)
  Determines if rows (`transpose` is `0`) or columns (`transpose` is `1`) are to be clustered.
- `nxgrid, nygrid` (default: `2, 1`)
  The number of cells horizontally and vertically in the rectangular grid on which the Self-Organizing Map is calculated.
- `inittau` (default: `0.02`)
  The initial value for the parameter τ that is used in the SOM algorithm. The default value for `inittau` is 0.02, which was used in Michael Eisen's Cluster/TreeView program.
- `niter` (default: `1`)
  The number of iterations to be performed.
- `dist` (default: `'e'`, Euclidean distance)
  Defines the distance function to be used (see 18.1).

This function returns the tuple `(clusterid, celldata)`:

- `clusterid`:
  An array with two columns, where the number of rows is equal to the number of items that were clustered. Each row contains the x and y coordinates of the cell in the rectangular SOM grid to which the item was assigned.

- `celldata`:
  An array with dimensions (`nxgrid`, `nygrid`, number of columns) if rows are being clustered, or (`nxgrid`, `nygrid`, number
  if columns are being clustered. Each element `[ix][iy]` of this array is a 1D vector containing the gene expression data for the centroid of the cluster in the grid cell with coordinates `[ix][iy]`.

Saving the clustering result

To save the clustering result, use

```
>>> record.save(jobname, geneclusters, expclusters)
```

where the following arguments are defined:

- `jobname`
  The string `jobname` is used as the base name for names of the files that are to be saved.

- `geneclusters`
  This argument describes the gene (row-wise) clustering result. In case of k-means clustering, this is a 1D array containing the number of the cluster each gene belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `geneclusters` is a `Tree` object.

- `expclusters`
  This argument describes the (column-wise) clustering result for the experimental conditions. In case of k-means clustering, this is a 1D array containing the number of the cluster each experimental condition belongs to. It can be calculated using `kcluster`. In case of hierarchical clustering, `expclusters` is a `Tree` object.

This method writes the text file `jobname.cdt`, `jobname.gtr`, `jobname.atr`, `jobname*.kgg`, and/or `jobname*.kag` for subsequent reading by the Java TreeView program. If `geneclusters` and `expclusters` are both `None`, this method only writes the text file `jobname.cdt`; this file can subsequently be read into a new `Record` object.

18.8 Example calculation

This is an example of a hierarchical clustering calculation, using single linkage clustering for genes and maximum linkage clustering for experimental conditions. As the Euclidean distance is being used for gene clustering, it is necessary to scale the node distances `genetree` such that they are all between zero and one. This is needed for the Java TreeView code to display the tree diagram correctly. To cluster the experimental conditions, the uncentered correlation is being used. No scaling is needed in this case, as the distances in `exptree` are already between zero and two.

The example data `cyano.txt` can be found in Biopython's `Tests/Cluster` subdirectory and is from the paper [20, Hihara *et al.*, 2001].

```
>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
...
```

```
>>> genetree = record.treecluster(method="s")
>>> genetree.scale()
>>> exptree = record.treecluster(dist="u", transpose=1)
>>> record.save("cyano_result", genetree, exptree)
```

This will create the files `cyano_result.cdt`, `cyano_result.gtr`, and `cyano_result.atr`.
Similarly, we can save a k-means clustering solution:

```
>>> from Bio import Cluster
>>> with open("cyano.txt") as handle:
...     record = Cluster.read(handle)
...
>>> (geneclusters, error, ifound) = record.kcluster(nclusters=5, npass=1000)
>>> (expclusters, error, ifound) = record.kcluster(nclusters=2, npass=100, transpose=1)
>>> record.save("cyano_result", geneclusters, expclusters)
```

This will create the files `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, and `cyano_result_K_A2.kag`.

Chapter 19

Graphics including GenomeDiagram

The `Bio.Graphics` module depends on the third party Python library ReportLab. Although focused on producing PDF files, ReportLab can also create encapsulated postscript (EPS) and (SVG) files. In addition to these vector based images, provided certain further dependencies such as the Python Imaging Library (PIL) are installed, ReportLab can also output bitmap images (including JPEG, PNG, GIF, BMP and PICT formats).

19.1 GenomeDiagram

19.1.1 Introduction

The `Bio.Graphics.GenomeDiagram` module was added to Biopython 1.50, having previously been available as a separate Python module dependent on Biopython. GenomeDiagram is described in the Bioinformatics journal publication by Pritchard et al. (2006) [35], which includes some examples images. There is a PDF copy of the old manual here, `http://biopython.org/DIST/docs/GenomeDiagram/userguide.pdf` which has some more examples.

As the name might suggest, GenomeDiagram was designed for drawing whole genomes, in particular prokaryotic genomes, either as linear diagrams (optionally broken up into fragments to fit better) or as circular wheel diagrams. Have a look at Figure 2 in Toth *et al.* (2006) [47] for a good example. It proved also well suited to drawing quite detailed figures for smaller genomes such as phage, plasmids or mitochondria, for example see Figures 1 and 2 in Van der Auwera *et al.* (2009) [49] (shown with additional manual editing).

This module is easiest to use if you have your genome loaded as a `SeqRecord` object containing lots of `SeqFeature` objects - for example as loaded from a GenBank file (see Chapters 4 and 5).

19.1.2 Diagrams, tracks, feature-sets and features

GenomeDiagram uses a nested set of objects. At the top level, you have a diagram object representing a sequence (or sequence region) along the horizontal axis (or circle). A diagram can contain one or more tracks, shown stacked vertically (or radially on circular diagrams). These will typically all have the same length and represent the same sequence region. You might use one track to show the gene locations, another to show regulatory regions, and a third track to show the GC percentage.

The most commonly used type of track will contain features, bundled together in feature-sets. You might choose to use one feature-set for all your CDS features, and another for tRNA features. This isn't required - they can all go in the same feature-set, but it makes it easier to update the properties of just selected features (e.g. make all the tRNA features red).

There are two main ways to build up a complete diagram. Firstly, the top down approach where you create a diagram object, and then using its methods add track(s), and use the track methods to add feature-

set(s), and use their methods to add the features. Secondly, you can create the individual objects separately (in whatever order suits your code), and then combine them.

19.1.3 A top down example

We're going to draw a whole genome from a `SeqRecord` object read in from a GenBank file (see Chapter 5). This example uses the pPCP1 plasmid from *Yersinia pestis biovar Microtus*, the file is included with the Biopython unit tests under the GenBank folder, or online NC_005816.gb from our website.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")
```

We're using a top down approach, so after loading in our sequence we next create an empty diagram, then add an (empty) track, and to that add an (empty) feature set:

```
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()
```

Now the fun part - we take each gene `SeqFeature` object in our `SeqRecord`, and use it to generate a feature on the diagram. We're going to color them blue, alternating between a dark blue and a light blue.

```
for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
```

Now we come to actually making the output file. This happens in two steps, first we call the `draw` method, which creates all the shapes using ReportLab objects. Then we call the `write` method which renders these to the requested file format. Note you can output in multiple file formats:

```
gd_diagram.draw(
    format="linear",
    orientation="landscape",
    pagesize="A4",
    fragments=4,
    start=0,
    end=len(record),
)
gd_diagram.write("plasmid_linear.pdf", "PDF")
gd_diagram.write("plasmid_linear.eps", "EPS")
gd_diagram.write("plasmid_linear.svg", "SVG")
```

Also, provided you have the dependencies installed, you can also do bitmaps, for example:

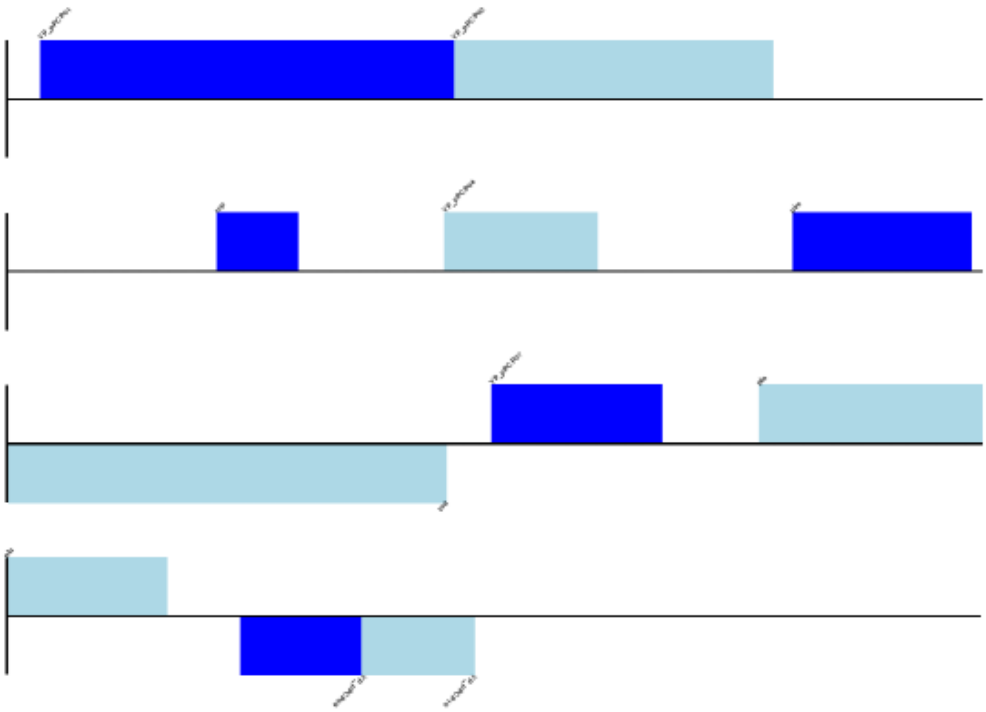


Figure 19.1: Simple linear diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1.

```
gd_diagram.write("plasmid_linear.png", "PNG")
```

The expected output is shown in Figure 19.1. Notice that the `fragments` argument which we set to four controls how many pieces the genome gets broken up into.

If you want to do a circular figure, then try this:

```
gd_diagram.draw(
    format="circular",
    circular=True,
    pagesize=(20 * cm, 20 * cm),
    start=0,
    end=len(record),
    circle_core=0.7,
)
gd_diagram.write("plasmid_circular.pdf", "PDF")
```

The expected output is shown in Figure 19.2. These figures are not very exciting, but we've only just got started.

19.1.4 A bottom up example

Now let's produce exactly the same figures, but using the bottom up approach. This means we create the different objects directly (and this can be done in almost any order) and then combine them.

```
from reportlab.lib import colors
from reportlab.lib.units import cm
```

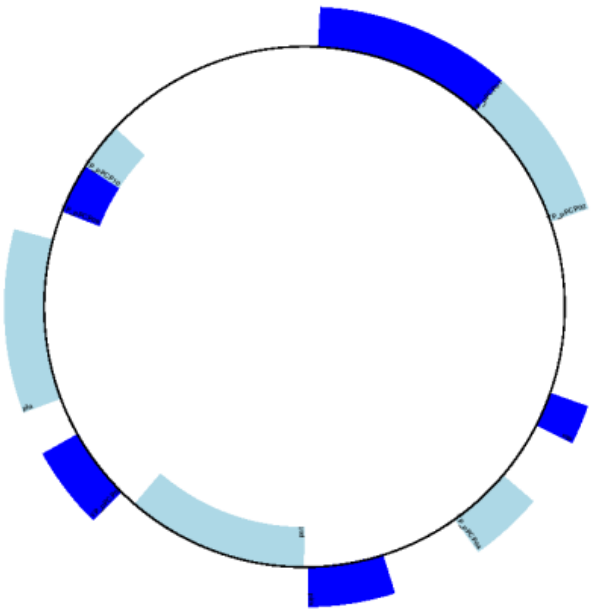


Figure 19.2: Simple circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1.

```
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")

# Create the feature set and its feature objects,
gd_feature_set = GenomeDiagram.FeatureSet()
for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
# (this for loop is the same as in the previous example)

# Create a track, and a diagram
gd_track_for_features = GenomeDiagram.Track(name="Annotated Features")
gd_diagram = GenomeDiagram.Diagram("Yersinia pestis biovar Microtus plasmid pPCP1")

# Now have to glue the bits together...
gd_track_for_features.add_set(gd_feature_set)
gd_diagram.add_track(gd_track_for_features, 1)
```

You can now call the `draw` and `write` methods as before to produce a linear or circular diagram, using the code at the end of the top-down example above. The figures should be identical.

19.1.5 Features without a SeqFeature

In the above example we used a `SeqRecord`'s `SeqFeature` objects to build our diagram (see also Section 4.3). Sometimes you won't have `SeqFeature` objects, but just the coordinates for a feature you want to draw. You have to create minimal `SeqFeature` object, but this is easy:

```
from Bio.SeqFeature import SeqFeature, SimpleLocation

my_seq_feature = SeqFeature(SimpleLocation(50, 100, strand=+1))
```

For strand, use `+1` for the forward strand, `-1` for the reverse strand, and `None` for both. Here is a short self contained example:

```
from Bio.SeqFeature import SeqFeature, SimpleLocation
from Bio.Graphics import GenomeDiagram
from reportlab.lib.units import cm

gdd = GenomeDiagram.Diagram("Test Diagram")
gdt_features = gdd.new_track(1, greytrack=False)
gds_features = gdt_features.new_set()

# Add three features to show the strand options,
feature = SeqFeature(SimpleLocation(25, 125, strand=+1))
gds_features.add_feature(feature, name="Forward", label=True)
feature = SeqFeature(SimpleLocation(150, 250, strand=None))
gds_features.add_feature(feature, name="Strandless", label=True)
feature = SeqFeature(SimpleLocation(275, 375, strand=-1))
gds_features.add_feature(feature, name="Reverse", label=True)

gdd.draw(format="linear", pagesize=(15 * cm, 4 * cm), fragments=1, start=0, end=400)
gdd.write("GD_labels_default.pdf", "pdf")
```

The output is shown at the top of Figure 19.3 (in the default feature color, pale green).

Notice that we have used the `name` argument here to specify the caption text for these features. This is discussed in more detail next.

19.1.6 Feature captions

Recall we used the following (where `feature` was a `SeqFeature` object) to add a feature to the diagram:

```
gd_feature_set.add_feature(feature, color=color, label=True)
```

In the example above the `SeqFeature` annotation was used to pick a sensible caption for the features. By default the following possible entries under the `SeqFeature` object's qualifiers dictionary are used: `gene`, `label`, `name`, `locus_tag`, and `product`. More simply, you can specify a name directly:

```
gd_feature_set.add_feature(feature, color=color, label=True, name="My Gene")
```

In addition to the caption text for each feature's label, you can also choose the font, position (this defaults to the start of the sigil, you can also choose the middle or at the end) and orientation (for linear diagrams only, where this defaults to rotated by 45 degrees):

```
# Large font, parallel with the track
gd_feature_set.add_feature(
    feature, label=True, color="green", label_size=25, label_angle=0
)

# Very small font, perpendicular to the track (towards it)
gd_feature_set.add_feature(
    feature,
    label=True,
    color="purple",
    label_position="end",
    label_size=4,
    label_angle=90,
)

# Small font, perpendicular to the track (away from it)
gd_feature_set.add_feature(
    feature,
    label=True,
    color="blue",
    label_position="middle",
    label_size=6,
    label_angle=-90,
)
```

Combining each of these three fragments with the complete example in the previous section should give something like the tracks in Figure 19.3.

We've not shown it here, but you can also set `label_color` to control the label's color (used in Section 19.1.9).

You'll notice the default font is quite small - this makes sense because you will usually be drawing many (small) features on a page, not just a few large ones as shown here.

19.1.7 Feature sigils

The examples above have all just used the default sigil for the feature, a plain box, which was all that was available in the last publicly released standalone version of GenomeDiagram. Arrow sigils were included when GenomeDiagram was added to Biopython 1.50:

```
# Default uses a BOX sigil
gd_feature_set.add_feature(feature)

# You can make this explicit:
gd_feature_set.add_feature(feature, sigil="BOX")

# Or opt for an arrow:
gd_feature_set.add_feature(feature, sigil="ARROW")
```

Biopython 1.61 added three more sigils,

```
# Box with corners cut off (making it an octagon)
gd_feature_set.add_feature(feature, sigil="OCTO")

# Box with jagged edges (useful for showing breaks in contains)
```

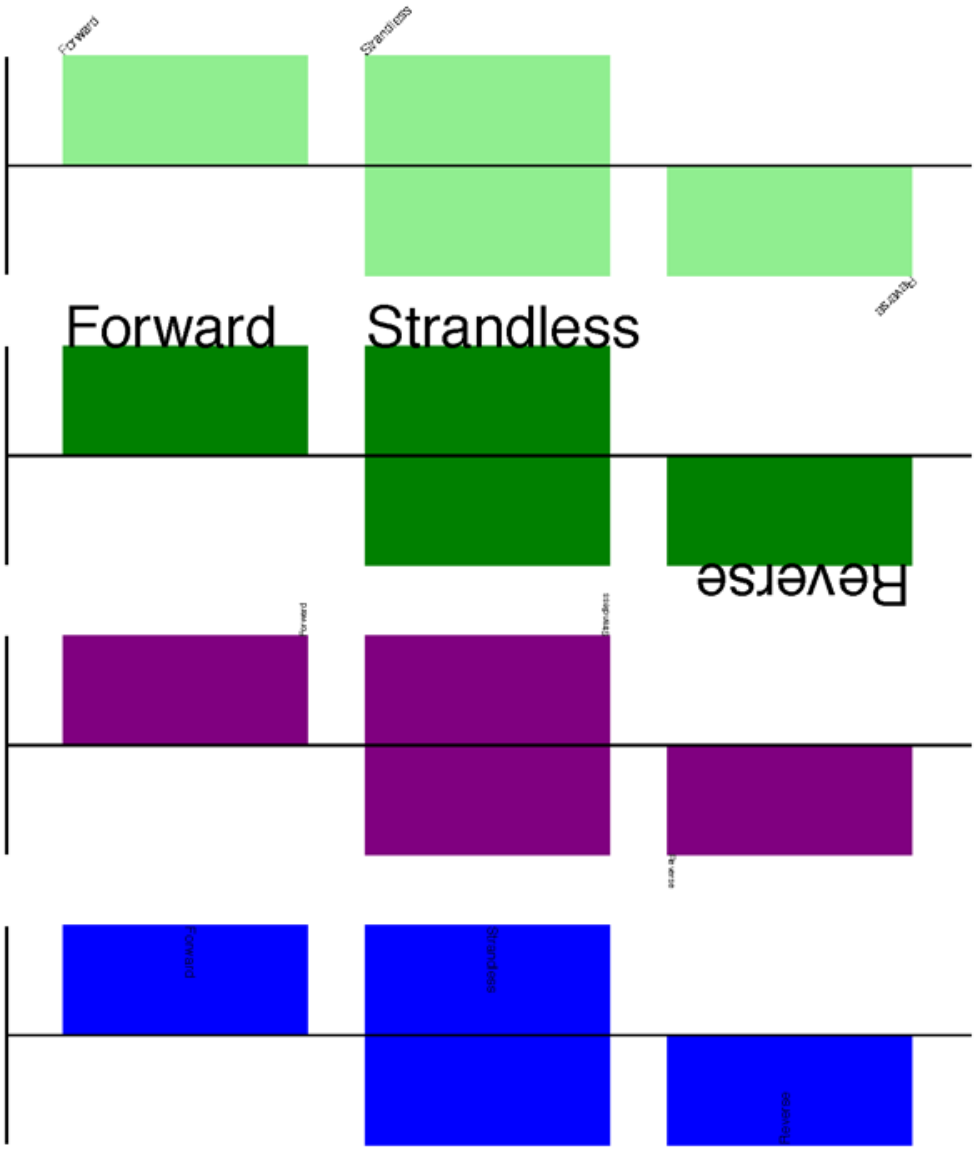


Figure 19.3: Simple GenomeDiagram showing label options. The top plot in pale green shows the default label settings (see Section 19.1.5) while the rest show variations in the label size, position and orientation (see Section 19.1.6).

```
gd_feature_set.add_feature(feature, sigil="JAGGY")

# Arrow which spans the axis with strand used only for direction
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

These are shown in Figure 19.4. Most sigils fit into a bounding box (as given by the default BOX sigil), either above or below the axis for the forward or reverse strand, or straddling it (double the height) for strand-less features. The BIGARROW sigil is different, always straddling the axis with the direction taken from the feature's stand.

19.1.8 Arrow sigils

We introduced the arrow sigils in the previous section. There are two additional options to adjust the shapes of the arrows, firstly the thickness of the arrow shaft, given as a proportion of the height of the bounding box:

```
# Full height shafts, giving pointed boxes:
gd_feature_set.add_feature(feature, sigil="ARROW", color="brown", arrowshaft_height=1.0)
# Or, thin shafts:
gd_feature_set.add_feature(feature, sigil="ARROW", color="teal", arrowshaft_height=0.2)
# Or, very thin shafts:
gd_feature_set.add_feature(
    feature, sigil="ARROW", color="darkgreen", arrowshaft_height=0.1
)
```

The results are shown in Figure 19.5.

Secondly, the length of the arrow head - given as a proportion of the height of the bounding box (defaulting to 0.5, or 50%):

```
# Short arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="blue", arrowhead_length=0.25)
# Or, longer arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="orange", arrowhead_length=1)
# Or, very very long arrow heads (i.e. all head, no shaft, so triangles):
gd_feature_set.add_feature(feature, sigil="ARROW", color="red", arrowhead_length=10000)
```

The results are shown in Figure 19.6.

Biopython 1.61 adds a new `BIGARROW` sigil which always straddles the axis, pointing left for the reverse strand or right otherwise:

```
# A large arrow straddling the axis:
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

All the shaft and arrow head options shown above for the `ARROW` sigil can be used for the `BIGARROW` sigil too.

19.1.9 A nice example

Now let's return to the pPCP1 plasmid from *Yersinia pestis biovar Microtus*, and the top down approach used in Section 19.1.3, but take advantage of the sigil options we've now discussed. This time we'll use arrows for the genes, and overlay them with strand-less features (as plain boxes) showing the position of some restriction digest sites.

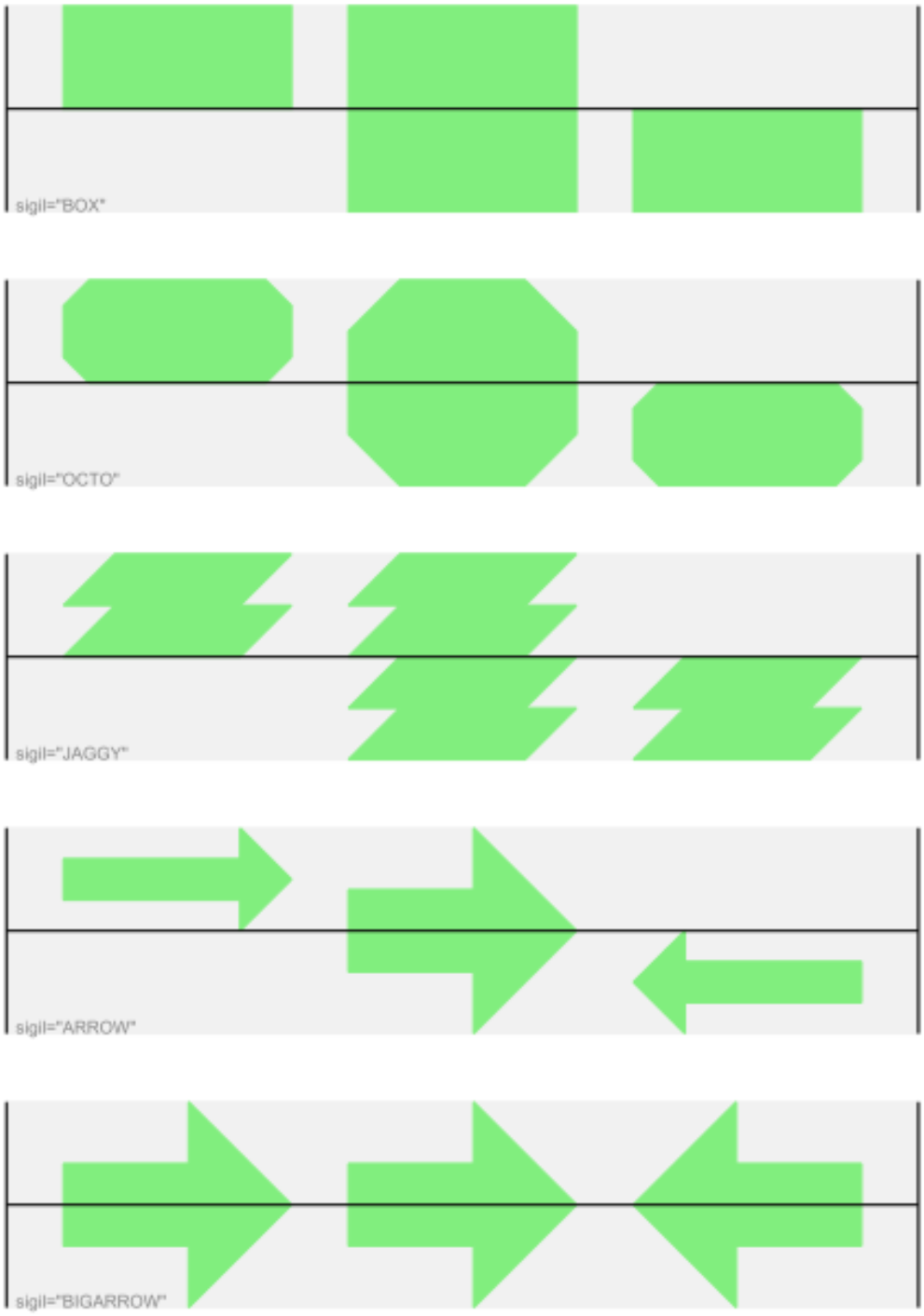


Figure 19.4: Simple GenomeDiagram showing different sigils (see Section 19.1.7)

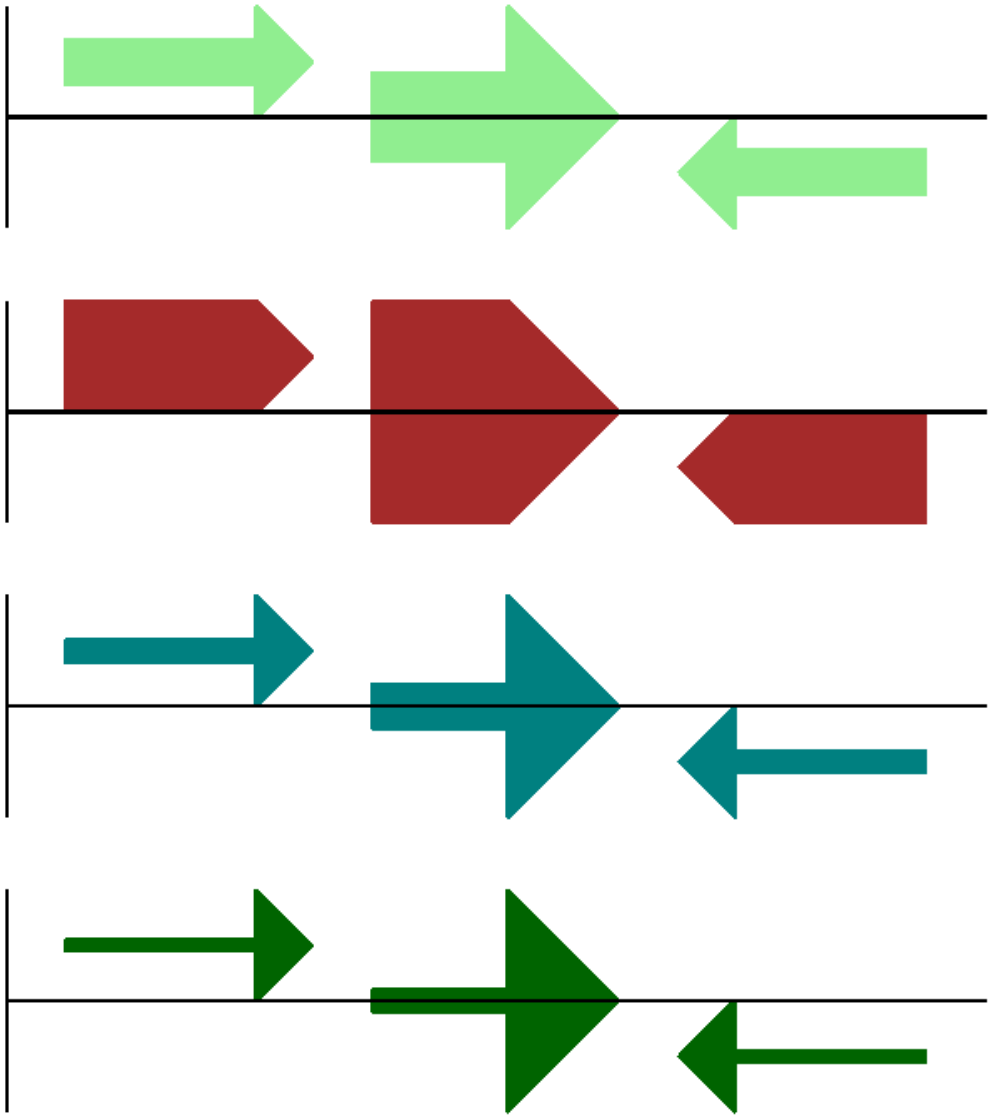


Figure 19.5: Simple GenomeDiagram showing arrow shaft options (see Section 19.1.8)

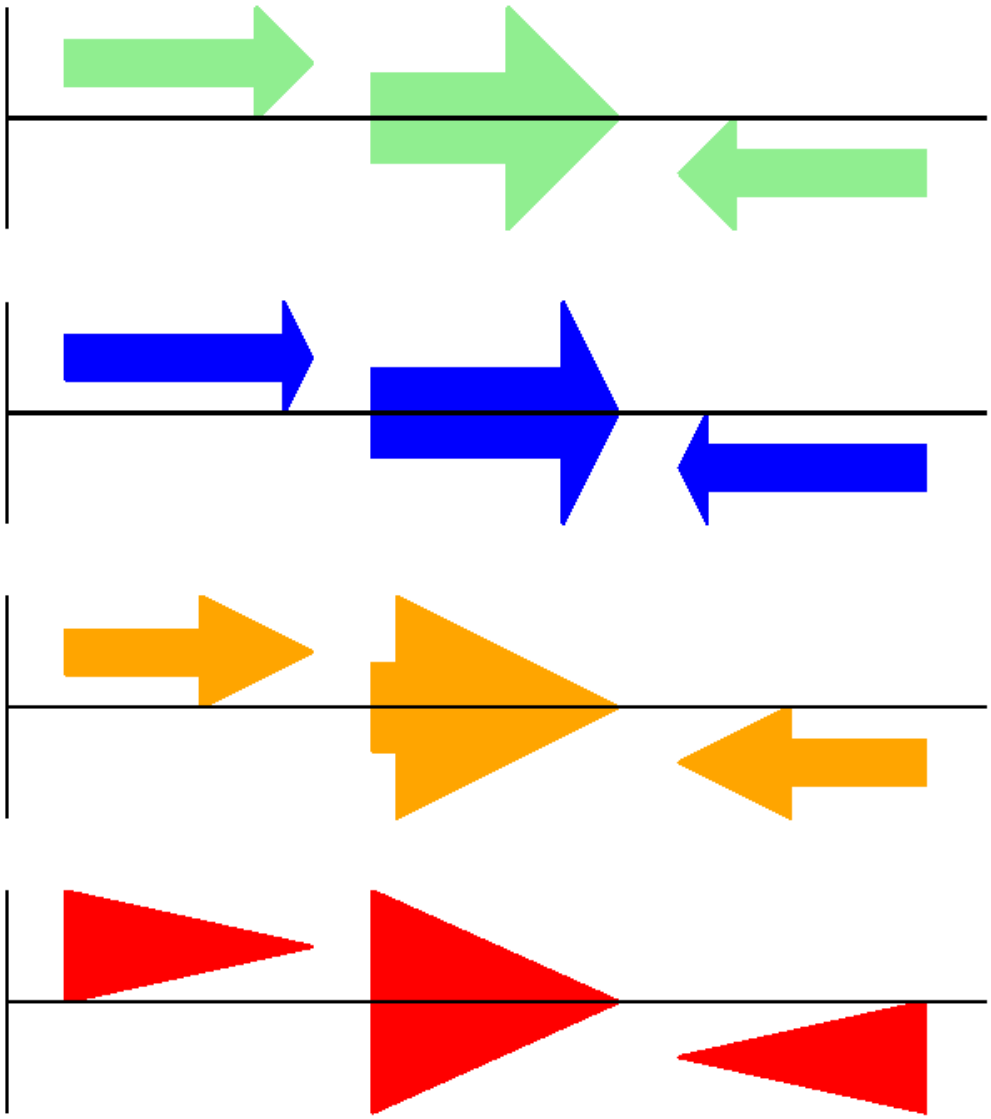


Figure 19.6: Simple GenomeDiagram showing arrow head options (see Section 19.1.8)

```
from reportlab.lib import colors
from reportlab.lib.units import cm
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
from Bio.SeqFeature import SeqFeature, SimpleLocation

record = SeqIO.read("NC_005816.gb", "genbank")

gd_diagram = GenomeDiagram.Diagram(record.id)
gd_track_for_features = gd_diagram.new_track(1, name="Annotated Features")
gd_feature_set = gd_track_for_features.new_set()

for feature in record.features:
    if feature.type != "gene":
        # Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(
        feature, sigil="ARROW", color=color, label=True, label_size=14, label_angle=0
    )

# I want to include some strandless features, so for an example
# will use EcoRI recognition sites etc.
for site, name, color in [
    ("GAATTC", "EcoRI", colors.green),
    ("CCCGGG", "SmaI", colors.orange),
    ("AAGCTT", "HindIII", colors.red),
    ("GGATCC", "BamHI", colors.purple),
]:
    index = 0
    while True:
        index = record.seq.find(site, start=index)
        if index == -1:
            break
        feature = SeqFeature(SimpleLocation(index, index + len(site)))
        gd_feature_set.add_feature(
            feature,
            color=color,
            name=name,
            label=True,
            label_size=10,
            label_color=color,
        )
        index += len(site)

gd_diagram.draw(format="linear", pagesize="A4", fragments=4, start=0, end=len(record))
gd_diagram.write("plasmid_linear_nice.pdf", "PDF")
gd_diagram.write("plasmid_linear_nice.eps", "EPS")
```

```
gd_diagram.write("plasmid_linear_nice.svg", "SVG")

gd_diagram.draw(
    format="circular",
    circular=True,
    pagesize=(20 * cm, 20 * cm),
    start=0,
    end=len(record),
    circle_core=0.5,
)
gd_diagram.write("plasmid_circular_nice.pdf", "PDF")
gd_diagram.write("plasmid_circular_nice.eps", "EPS")
gd_diagram.write("plasmid_circular_nice.svg", "SVG")
```

The expected output is shown in Figures 19.7 and 19.8.

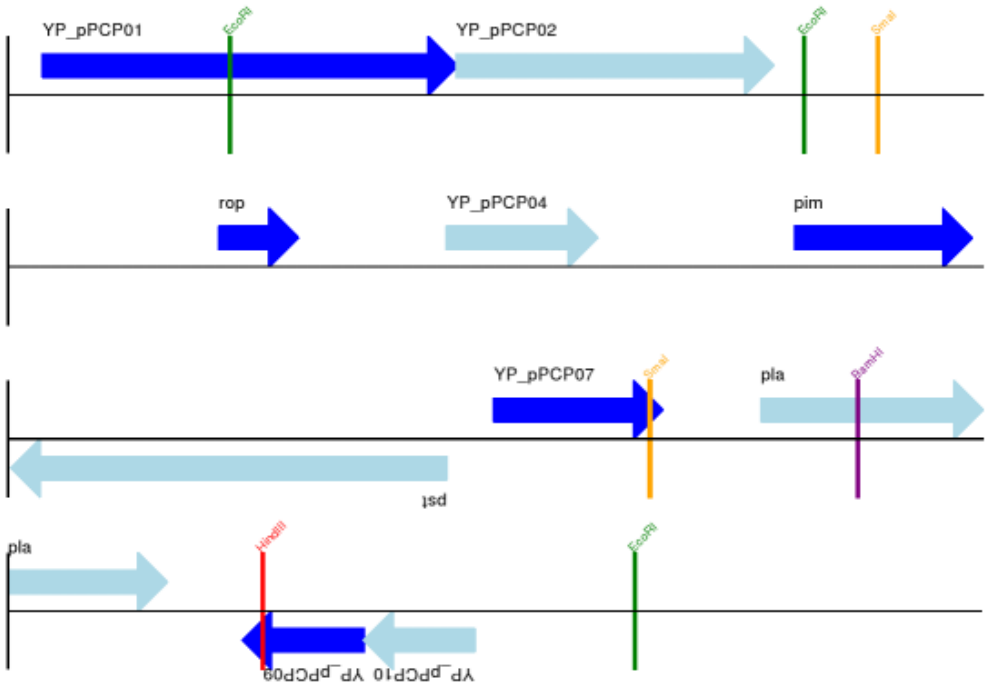


Figure 19.7: Linear diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section 19.1.9).

19.1.10 Multiple tracks

All the examples so far have used a single track, but you can have more than one track – for example show the genes on one, and repeat regions on another. In this example we're going to show three phage genomes side by side to scale, inspired by Figure 6 in Proux *et al.* (2002) [36]. We'll need the GenBank files for the following three phage:

- `NC_002703` – Lactococcus phage Tuc2009, complete genome (38347 bp)

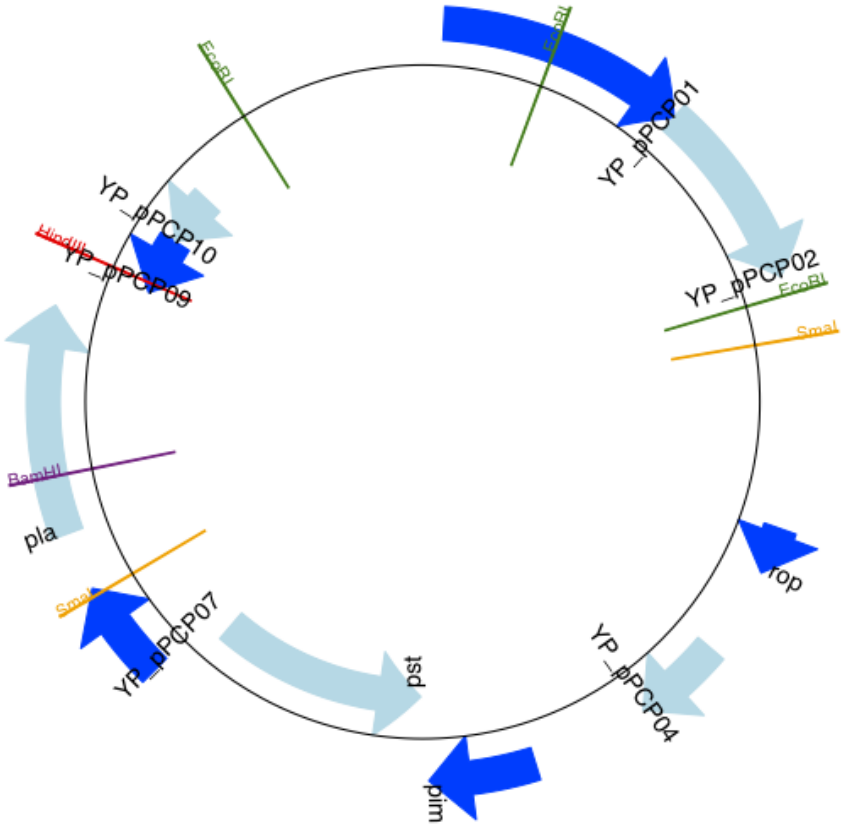


Figure 19.8: Circular diagram for *Yersinia pestis biovar Microtus* plasmid pPCP1 showing selected restriction digest sites (see Section 19.1.9).

- `AF323668` – Bacteriophage bIL285, complete genome (35538 bp)
- `NC_003212` – *Listeria innocua* Clip11262, complete genome, of which we are focussing only on integrated prophage 5 (similar length).

You can download these using Entrez if you like, see Section 12.6 for more details. For the third record we've worked out where the phage is integrated into the genome, and slice the record to extract it (with the features preserved, see Section 4.7), and must also reverse complement to match the orientation of the first two phage (again preserving the features, see Section 4.9):

```
from Bio import SeqIO

A_rec = SeqIO.read("NC_002703.gbk", "gb")
B_rec = SeqIO.read("AF323668.gbk", "gb")
C_rec = SeqIO.read("NC_003212.gbk", "gb")[2587879:2625807].reverse_complement(name=True)
```

The figure we are imitating used different colors for different gene functions. One way to do this is to edit the GenBank file to record color preferences for each feature - something Sanger's Artemis editor does, and which GenomeDiagram should understand. Here however, we'll just hard code three lists of colors.

Note that the annotation in the GenBank files doesn't exactly match that shown in Proux *et al.*, they have drawn some unannotated genes.

```
from reportlab.lib.colors import (
    red,
    grey,
    orange,
    green,
    brown,
    blue,
    lightblue,
    purple,
)

A_colors = (
    [red] * 5
    + [grey] * 7
    + [orange] * 2
    + [grey] * 2
    + [orange]
    + [grey] * 11
    + [green] * 4
    + [grey]
    + [green] * 2
    + [grey, green]
    + [brown] * 5
    + [blue] * 4
    + [lightblue] * 5
    + [grey, lightblue]
    + [purple] * 2
    + [grey]
)
B_colors = (
    [red] * 6
```

```
    + [grey] * 8
    + [orange] * 2
    + [grey]
    + [orange]
    + [grey] * 21
    + [green] * 5
    + [grey]
    + [brown] * 4
    + [blue] * 3
    + [lightblue] * 3
    + [grey] * 5
    + [purple] * 2
)
C_colors = (
    [grey] * 30
    + [green] * 5
    + [brown] * 4
    + [blue] * 2
    + [grey, blue]
    + [lightblue] * 2
    + [grey] * 5
)
```

Now to draw them – this time we add three tracks to the diagram, and also notice they are given different start/end values to reflect their different lengths (this requires Biopython 1.59 or later).

```
from Bio.Graphics import GenomeDiagram

name = "Proux Fig 6"
gd_diagram = GenomeDiagram.Diagram(name)
max_len = 0
for record, gene_colors in zip([A_rec, B_rec, C_rec], [A_colors, B_colors, C_colors]):
    max_len = max(max_len, len(record))
    gd_track_for_features = gd_diagram.new_track(
        1, name=record.name, greytrack=True, start=0, end=len(record)
    )
    gd_feature_set = gd_track_for_features.new_set()

    i = 0
    for feature in record.features:
        if feature.type != "gene":
            # Exclude this feature
            continue
        gd_feature_set.add_feature(
            feature,
            sigil="ARROW",
            color=gene_colors[i],
            label=True,
            name=str(i + 1),
            label_position="start",
            label_size=6,
            label_angle=0,
```

```
        )
        i += 1

gd_diagram.draw(format="linear", pagesize="A4", fragments=1, start=0, end=max_len)
gd_diagram.write(name + ".pdf", "PDF")
gd_diagram.write(name + ".eps", "EPS")
gd_diagram.write(name + ".svg", "SVG")
```

The expected output is shown in Figure 19.9. I did wonder why in the original manuscript there were no

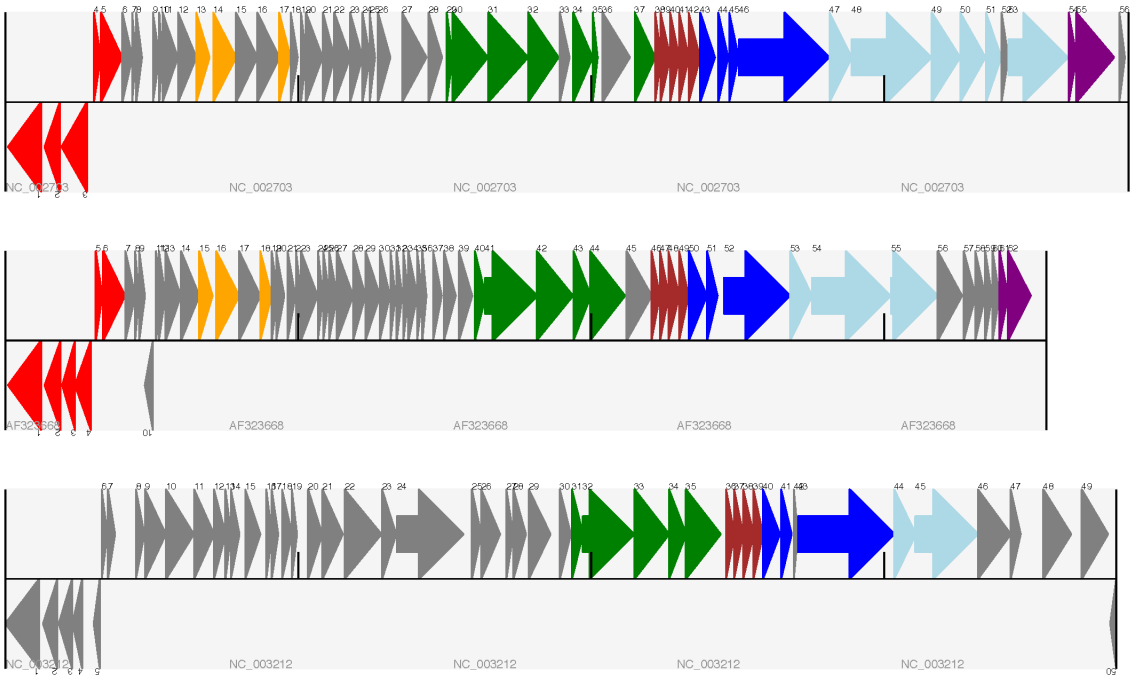


Figure 19.9: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) (see Section 19.1.10).

red or orange genes marked in the bottom phage. Another important point is here the phage are shown with different lengths - this is because they are all drawn to the same scale (they *are* different lengths).

The key difference from the published figure is they have color-coded links between similar proteins – which is what we will do in the next section.

19.1.11 Cross-Links between tracks

Biopython 1.59 added the ability to draw cross links between tracks - both simple linear diagrams as we will show here, but also linear diagrams split into fragments and circular diagrams.

Continuing the example from the previous section inspired by Figure 6 from Proux *et al.* 2002 [36], we would need a list of cross links between pairs of genes, along with a score or color to use. Realistically you might extract this from a BLAST file computationally, but here I have manually typed them in.

My naming convention continues to refer to the three phage as A, B and C. Here are the links we want to show between A and B, given as a list of tuples (percentage similarity score, gene in A, gene in B).

```
# Tuc2009 (NC_002703) vs bIL285 (AF323668)
A_vs_B = [
    (99, "Tuc2009_01", "int"),
    (33, "Tuc2009_03", "orf4"),
    (94, "Tuc2009_05", "orf6"),
    (100, "Tuc2009_06", "orf7"),
    (97, "Tuc2009_07", "orf8"),
    (98, "Tuc2009_08", "orf9"),
    (98, "Tuc2009_09", "orf10"),
    (100, "Tuc2009_10", "orf12"),
    (100, "Tuc2009_11", "orf13"),
    (94, "Tuc2009_12", "orf14"),
    (87, "Tuc2009_13", "orf15"),
    (94, "Tuc2009_14", "orf16"),
    (94, "Tuc2009_15", "orf17"),
    (88, "Tuc2009_17", "rusA"),
    (91, "Tuc2009_18", "orf20"),
    (93, "Tuc2009_19", "orf22"),
    (71, "Tuc2009_20", "orf23"),
    (51, "Tuc2009_22", "orf27"),
    (97, "Tuc2009_23", "orf28"),
    (88, "Tuc2009_24", "orf29"),
    (26, "Tuc2009_26", "orf38"),
    (19, "Tuc2009_46", "orf52"),
    (77, "Tuc2009_48", "orf54"),
    (91, "Tuc2009_49", "orf55"),
    (95, "Tuc2009_52", "orf60"),
]
```

Likewise for B and C:

```
# bIL285 (AF323668) vs Listeria innocua prophage 5 (in NC_003212)
B_vs_C = [
    (42, "orf39", "lin2581"),
    (31, "orf40", "lin2580"),
    (49, "orf41", "lin2579"),  # terL
    (54, "orf42", "lin2578"),  # portal
    (55, "orf43", "lin2577"),  # protease
    (33, "orf44", "lin2576"),  # mhp
    (51, "orf46", "lin2575"),
    (33, "orf47", "lin2574"),
    (40, "orf48", "lin2573"),
    (25, "orf49", "lin2572"),
    (50, "orf50", "lin2571"),
    (48, "orf51", "lin2570"),
    (24, "orf52", "lin2568"),
```

```
    (30, "orf53", "lin2567"),
    (28, "orf54", "lin2566"),
]
```

For the first and last phage these identifiers are locus tags, for the middle phage there are no locus tags so I've used gene names instead. The following little helper function lets us lookup a feature using either a locus tag or gene name:

```
def get_feature(features, id, tags=["locus_tag", "gene"]):
    """Search list of SeqFeature objects for an identifier under the given tags."""
    for f in features:
        for key in tags:
            # tag may not be present in this feature
            for x in f.qualifiers.get(key, []):
                if x == id:
                    return f
    raise KeyError(id)
```

We can now turn those list of identifier pairs into SeqFeature pairs, and thus find their location coordinates. We can now add all that code and the following snippet to the previous example (just before the `gd_diagram.draw(...)` line – see the finished example script Proux_et_al_2002_Figure_6.py included in the `Doc/examples` folder of the Biopython source code) to add cross links to the figure:

```
from Bio.Graphics.GenomeDiagram import CrossLink
from reportlab.lib import colors

# Note it might have been clearer to assign the track numbers explicitly...
for rec_X, tn_X, rec_Y, tn_Y, X_vs_Y in [
    (A_rec, 3, B_rec, 2, A_vs_B),
    (B_rec, 2, C_rec, 1, B_vs_C),
]:
    track_X = gd_diagram.tracks[tn_X]
    track_Y = gd_diagram.tracks[tn_Y]
    for score, id_X, id_Y in X_vs_Y:
        feature_X = get_feature(rec_X.features, id_X)
        feature_Y = get_feature(rec_Y.features, id_Y)
        color = colors.linearlyInterpolatedColor(
            colors.white, colors.firebrick, 0, 100, score
        )
        link_xy = CrossLink(
            (track_X, feature_X.location.start, feature_X.location.end),
            (track_Y, feature_Y.location.start, feature_Y.location.end),
            color,
            colors.lightgrey,
        )
        gd_diagram.cross_track_links.append(link_xy)
```

There are several important pieces to this code. First the `GenomeDiagram` object has a `cross_track_links` attribute which is just a list of `CrossLink` objects. Each `CrossLink` object takes two sets of track-specific coordinates (here given as tuples, you can alternatively use a `GenomeDiagram.Feature` object instead). You can optionally supply a color, border color, and say if this link should be drawn flipped (useful for showing inversions).

You can also see how we turn the BLAST percentage identity score into a color, interpolating between white (0%) and a dark red (100%). In this example we don't have any problems with overlapping cross-links. One way to tackle that is to use transparency in ReportLab, by using colors with their alpha channel set. However, this kind of shaded color scheme combined with overlap transparency would be difficult to interpret. The expected output is shown in Figure 19.10.

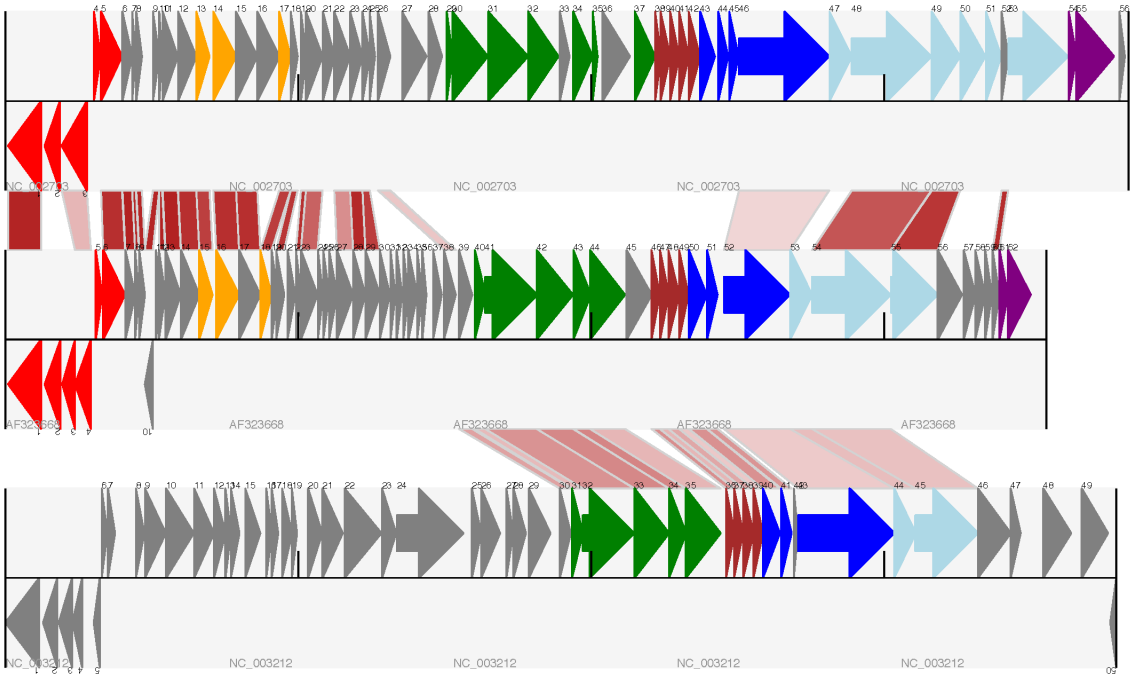


Figure 19.10: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus basic cross-links shaded by percentage identity (see Section 19.1.11).

There is still a lot more that can be done within Biopython to help improve this figure. First of all, the cross links in this case are between proteins which are drawn in a strand specific manor. It can help to add a background region (a feature using the 'BOX' sigil) on the feature track to extend the cross link. Also, we could reduce the vertical height of the feature tracks to allocate more to the links instead – one way to do that is to allocate space for empty tracks. Furthermore, in cases like this where there are no large gene overlaps, we can use the axis-straddling `BIGARROW` sigil, which allows us to further reduce the vertical space needed for the track. These improvements are demonstrated in the example script Proux_et_al_2002_Figure_6.py included in the `Doc/examples` folder of the Biopython source code. The expected output is shown in Figure 19.11.

Beyond that, finishing touches you might want to do manually in a vector image editor include fine tuning the placement of gene labels, and adding other custom annotation such as highlighting particular regions.

Although not really necessary in this example since none of the cross-links overlap, using a transparent color in ReportLab is a very useful technique for superimposing multiple links. However, in this case a

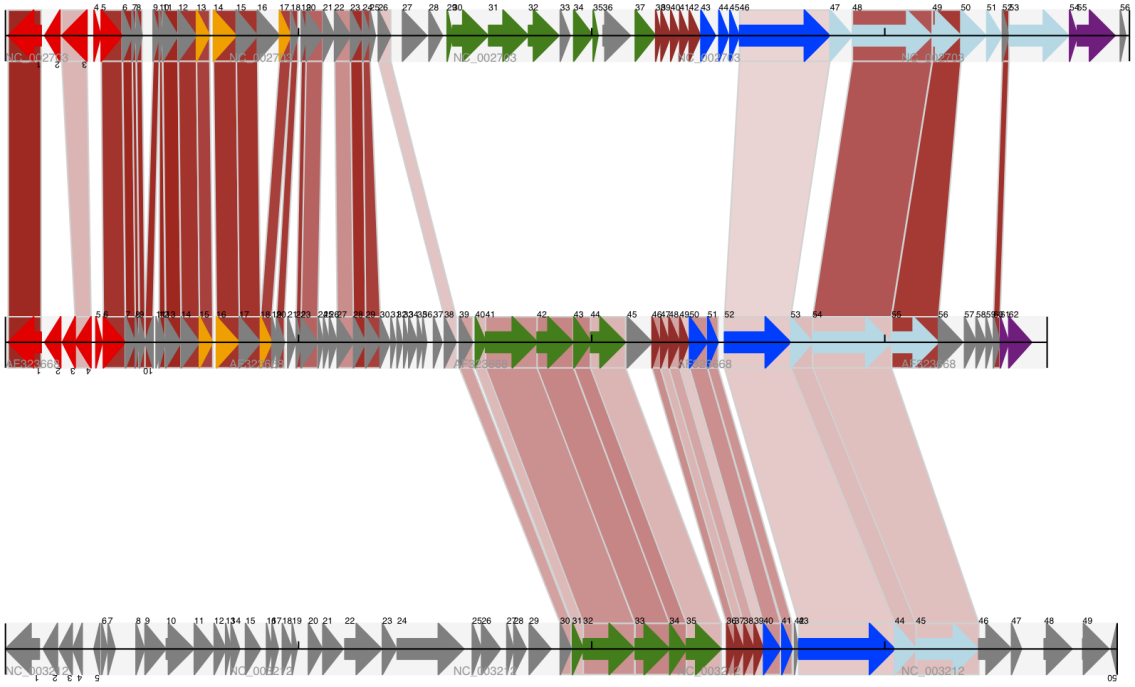


Figure 19.11: Linear diagram with three tracks for Lactococcus phage Tuc2009 (NC_002703), bacteriophage bIL285 (AF323668), and prophage 5 from *Listeria innocua* Clip11262 (NC_003212) plus cross-links shaded by percentage identity (see Section 19.1.11).

shaded color scheme should be avoided.

19.1.12 Further options

You can control the tick marks to show the scale – after all every graph should show its units, and the number of the grey-track labels.

Also, we have only used the `FeatureSet` so far. GenomeDiagram also has a `GraphSet` which can be used for show line graphs, bar charts and heat plots (e.g. to show plots of GC% on a track parallel to the features).

These options are not covered here yet, so for now we refer you to the User Guide (PDF) included with the standalone version of GenomeDiagram (but please read the next section first), and the docstrings.

19.1.13 Converting old code

If you have old code written using the standalone version of GenomeDiagram, and you want to switch it over to using the new version included with Biopython then you will have to make a few changes - most importantly to your import statements.

Also, the older version of GenomeDiagram used only the UK spellings of color and center (colour and centre). You will need to change to the American spellings, although for several years the Biopython version of GenomeDiagram supported both.

For example, if you used to have:

```
from GenomeDiagram import GDFeatureSet, GDDiagram

gdd = GDDiagram("An example")
...
```

you could just switch the import statements like this:

```
from Bio.Graphics.GenomeDiagram import FeatureSet as GDFeatureSet, Diagram as GDDiagram

gdd = GDDiagram("An example")
...
```

and hopefully that should be enough. In the long term you might want to switch to the new names, but you would have to change more of your code:

```
from Bio.Graphics.GenomeDiagram import FeatureSet, Diagram

gdd = Diagram("An example")
...
```

or:

```
from Bio.Graphics import GenomeDiagram

gdd = GenomeDiagram.Diagram("An example")
...
```

If you run into difficulties, please ask on the Biopython mailing list for advice. One catch is that we have not included the old module `GenomeDiagram.GDUtilities` yet. This included a number of GC% related functions, which will probably be merged under `Bio.SeqUtils` later on.

19.2 Chromosomes

The `Bio.Graphics.BasicChromosome` module allows drawing of chromosomes. There is an example in Jupe *et al.* (2012) [22] (open access) using colors to highlight different gene families.

19.2.1 Simple Chromosomes

Here is a very simple example - for which we'll use *Arabidopsis thaliana*.

You can skip this bit, but first I downloaded the five sequenced chromosomes as five individual FASTA files from the NCBI's FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/ and then parsed them with `Bio.SeqIO` to find out their lengths. You could use the GenBank files for this (and the next example uses those for plotting features), but if all you want is the length it is faster to use the FASTA files for the whole chromosomes:

```
from Bio import SeqIO

entries = [
    ("Chr I", "CHR_I/NC_003070.fna"),
    ("Chr II", "CHR_II/NC_003071.fna"),
    ("Chr III", "CHR_III/NC_003074.fna"),
    ("Chr IV", "CHR_IV/NC_003075.fna"),
    ("Chr V", "CHR_V/NC_003076.fna"),
]
for name, filename in entries:
    record = SeqIO.read(filename, "fasta")
    print(name, len(record))
```

This gave the lengths of the five chromosomes, which we'll now use in the following short demonstration of the `BasicChromosome` module:

```
from reportlab.lib.units import cm
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", 30432563),
    ("Chr II", 19705359),
    ("Chr III", 23470805),
    ("Chr IV", 18585042),
    ("Chr V", 26992728),
]

max_len = 30432563  # Could compute this from the entries dict
telomere_length = 1000000  # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm)  # A4 landscape

for name, length in entries:
    cur_chromosome = BasicChromosome.Chromosome(name)
    # Set the scale to the MAXIMUM length plus the two telomeres in bp,
    # want the same scale used on all five chromosomes so they can be
    # compared to each other
    cur_chromosome.scale_num = max_len + 2 * telomere_length
```

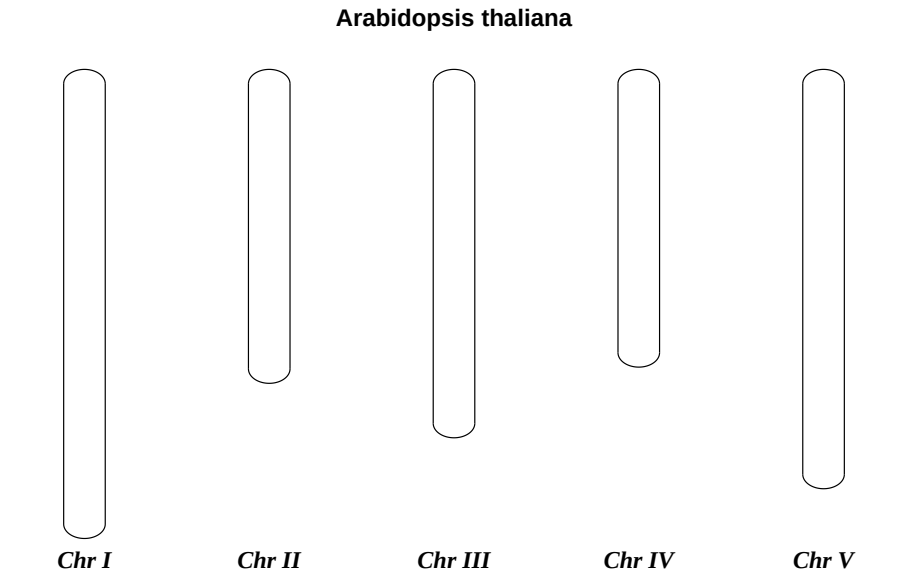


Figure 19.12: Simple chromosome diagram for *Arabidopsis thaliana*.

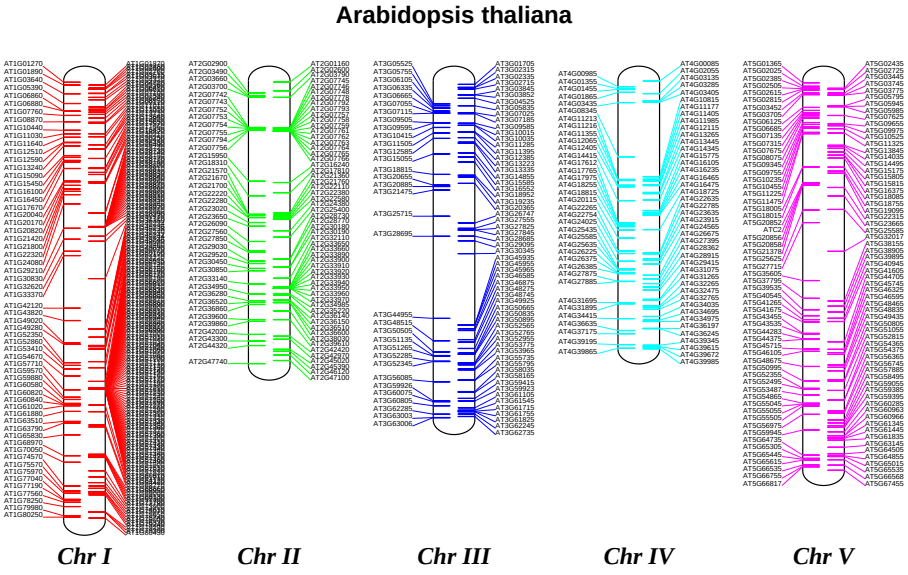


Figure 19.13: Chromosome diagram for *Arabidopsis thaliana* showing tRNA genes.

```
    # Add an opening telomere
    start = BasicChromosome.TelomereSegment()
    start.scale = telomere_length
    cur_chromosome.add(start)

    # Add a body - using bp as the scale length here.
    body = BasicChromosome.ChromosomeSegment()
    body.scale = length
    cur_chromosome.add(body)

    # Add a closing telomere
    end = BasicChromosome.TelomereSegment(inverted=True)
    end.scale = telomere_length
    cur_chromosome.add(end)

    # This chromosome is done
    chr_diagram.add(cur_chromosome)

chr_diagram.draw("simple_chrom.pdf", "Arabidopsis thaliana")
```

This should create a very simple PDF file, shown in Figure 19.12. This example is deliberately short and sweet. The next example shows the location of features of interest.

19.2.2 Annotated Chromosomes

Continuing from the previous example, let's also show the tRNA genes. We'll get their locations by parsing the GenBank files for the five *Arabidopsis thaliana* chromosomes. You'll need to download these files from the NCBI FTP site ftp://ftp.ncbi.nlm.nih.gov/genomes/archive/old_refseq/Arabidopsis_thaliana/, and preserve the subdirectory names or edit the paths below:

```
from reportlab.lib.units import cm
from Bio import SeqIO
from Bio.Graphics import BasicChromosome

entries = [
    ("Chr I", "CHR_I/NC_003070.gbk"),
    ("Chr II", "CHR_II/NC_003071.gbk"),
    ("Chr III", "CHR_III/NC_003074.gbk"),
    ("Chr IV", "CHR_IV/NC_003075.gbk"),
    ("Chr V", "CHR_V/NC_003076.gbk"),
]

max_len = 30432563  # Could compute this from the entries dict
telomere_length = 1000000  # For illustration

chr_diagram = BasicChromosome.Organism()
chr_diagram.page_size = (29.7 * cm, 21 * cm)  # A4 landscape

for index, (name, filename) in enumerate(entries):
    record = SeqIO.read(filename, "genbank")
    length = len(record)
```

```
    features = [f for f in record.features if f.type == "tRNA"]
    # Record an Artemis style integer color in the feature's qualifiers,
    # 1 = Black, 2 = Red, 3 = Green, 4 = blue, 5 =cyan, 6 = purple
    for f in features:
        f.qualifiers["color"] = [index + 2]

    cur_chromosome = BasicChromosome.Chromosome(name)
    # Set the scale to the MAXIMUM length plus the two telomeres in bp,
    # want the same scale used on all five chromosomes so they can be
    # compared to each other
    cur_chromosome.scale_num = max_len + 2 * telomere_length

    # Add an opening telomere
    start = BasicChromosome.TelomereSegment()
    start.scale = telomere_length
    cur_chromosome.add(start)

    # Add a body - again using bp as the scale length here.
    body = BasicChromosome.AnnotatedChromosomeSegment(length, features)
    body.scale = length
    cur_chromosome.add(body)

    # Add a closing telomere
    end = BasicChromosome.TelomereSegment(inverted=True)
    end.scale = telomere_length
    cur_chromosome.add(end)

    # This chromosome is done
    chr_diagram.add(cur_chromosome)

chr_diagram.draw("tRNA_chrom.pdf", "Arabidopsis thaliana")
```

It might warn you about the labels being too close together - have a look at the forward strand (right hand side) of Chr I, but it should create a colorful PDF file, shown in Figure 19.12.

Chapter 20

KEGG

KEGG (https://www.kegg.jp/) is a database resource for understanding high-level functions and utilities of the biological system, such as the cell, the organism and the ecosystem, from molecular-level information, especially large-scale molecular datasets generated by genome sequencing and other high-throughput experimental technologies.

Please note that the KEGG parser implementation in Biopython is incomplete. While the KEGG website indicates many flat file formats, only parsers and writers for compound, enzyme, and map are currently implemented. However, a generic parser is implemented to handle the other formats.

20.1 Parsing KEGG records

Parsing a KEGG record is as simple as using any other file format parser in Biopython. (Before running the following codes, please open http://rest.kegg.jp/get/ec:5.4.2.2 with your web browser and save it as `ec_5.4.2.2.txt`.)

```
>>> from Bio.KEGG import Enzyme
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Alternatively, if the input KEGG file has exactly one entry, you can use `read`:

```
>>> from Bio.KEGG import Enzyme
>>> record = Enzyme.read(open("ec_5.4.2.2.txt"))
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

The following section will shows how to download the above enzyme using the KEGG api as well as how to use the generic parser with data that does not have a custom parser implemented.

20.2 Querying the KEGG API

Biopython has full support for the querying of the KEGG api. Querying all KEGG endpoints are supported; all methods documented by KEGG (https://www.kegg.jp/kegg/rest/keggapi.html) are supported. The

interface has some validation of queries which follow rules defined on the KEGG site. However, invalid queries which return a 400 or 404 must be handled by the user.

First, here is how to extend the above example by downloading the relevant enzyme and passing it through the Enzyme parser.

```
>>> from Bio.KEGG import REST
>>> from Bio.KEGG import Enzyme
>>> request = REST.kegg_get("ec:5.4.2.2")
>>> open("ec_5.4.2.2.txt", "w").write(request.read())
>>> records = Enzyme.parse(open("ec_5.4.2.2.txt"))
>>> record = list(records)[0]
>>> record.classname
['Isomerases;', 'Intramolecular transferases;', 'Phosphotransferases (phosphomutases)']
>>> record.entry
'5.4.2.2'
```

Now, here's a more realistic example which shows a combination of querying the KEGG API. This will demonstrate how to extract a unique set of all human pathway gene symbols which relate to DNA repair. The steps that need to be taken to do so are as follows. First, we need to get a list of all human pathways. Secondly, we need to filter those for ones which relate to "repair". Lastly, we need to get a list of all the gene symbols in all repair pathways.

```
from Bio.KEGG import REST

human_pathways = REST.kegg_list("pathway", "hsa").read()

# Filter all human pathways for repair pathways
repair_pathways = []
for line in human_pathways.rstrip().split("\n"):
    entry, description = line.split("\t")
    if "repair" in description:
        repair_pathways.append(entry)

# Get the genes for pathways and add them to a list
repair_genes = []
for pathway in repair_pathways:
    pathway_file = REST.kegg_get(pathway).read()  # query and read each pathway

    # iterate through each KEGG pathway file, keeping track of which section
    # of the file we're in, only read the gene in each pathway
    current_section = None
    for line in pathway_file.rstrip().split("\n"):
        section = line[:12].strip()  # section names are within 12 columns
        if not section == "":
            current_section = section

        if current_section == "GENE":
            gene_identifiers, gene_description = line[12:].split("; ")
            gene_id, gene_symbol = gene_identifiers.split()

            if not gene_symbol in repair_genes:
                repair_genes.append(gene_symbol)
```

```
print(
    "There are %d repair pathways and %d repair genes. The genes are:"
    % (len(repair_pathways), len(repair_genes))
)
print(", ".join(repair_genes))
```

The KEGG API wrapper is compatible with all endpoints. Usage is essentially replacing all slashes in the url with commas and using that list as arguments to the corresponding method in the KEGG module. Here are a few examples from the api documentation (https://www.kegg.jp/kegg/docs/keggapi.html).

```
/list/hsa:10458+ece:Z5100                -> REST.kegg_list(["hsa:10458", "ece:Z5100"])
/find/compound/300-310/mol_weight        -> REST.kegg_find("compound", "300-310", "mol_weight")
/get/hsa:10458+ece:Z5100/aaseq           -> REST.kegg_get(["hsa:10458", "ece:Z5100"], "aaseq")
```

Chapter 21

Bio.phenotype: analyze phenotypic data

This chapter gives an overview of the functionalities of the `Bio.phenotype` package included in Biopython. The scope of this package is the analysis of phenotypic data, which means parsing and analyzing growth measurements of cell cultures. In its current state the package is focused on the analysis of high-throughput phenotypic experiments produced by the Phenotype Microarray technology, but future developments may include other platforms and formats.

21.1 Phenotype Microarrays

The Phenotype Microarray is a technology that measures the metabolism of bacterial and eukaryotic cells on roughly 2000 chemicals, divided in twenty 96-well plates. The technology measures the reduction of a tetrazolium dye by NADH, whose production by the cell is used as a proxy for cell metabolism; color development due to the reduction of this dye is typically measured once every 15 minutes. When cells are grown in a media that sustains cell metabolism, the recorded phenotypic data resembles a sigmoid growth curve, from which a series of growth parameters can be retrieved.

21.1.1 Parsing Phenotype Microarray data

The `Bio.phenotype` package can parse two different formats of Phenotype Microarray data: the CSV (comma separated values) files produced by the machine's proprietary software and JSON files produced by analysis software, like opm or DuctApe. The parser will return one or a generator of PlateRecord objects, depending on whether the read or parse method is being used. You can test the parse function by using the `Plates.csv` file provided with the Biopython source code.

```
>>> from Bio import phenotype
>>> for record in phenotype.parse("Plates.csv", "pm-csv"):
...     print("%s %i" % (record.id, len(record)))
...
PM01 96
PM01 96
PM09 96
PM09 96
```

The parser returns a series of PlateRecord objects, each one containing a series of WellRecord objects (holding each well's experimental data) arranged in 8 rows and 12 columns; each row is indicated by a

uppercase character from A to H, while columns are indicated by a two digit number, from 01 to 12. There are several ways to access WellRecord objects from a PlateRecord objects:

Well identifier If you know the well identifier (row + column identifiers) you can access the desired well directly.

```
>>> record["A02"]
WellRecord('(0.0, 12.0), (0.25, 18.0), (0.5, 27.0), (0.75, 35.0), (1.0, 37.0), ..., (71.75, 143.0)'
```

Well plate coordinates The same well can be retrieved by using the row and columns numbers (0-based index).

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> print(record[0, 1].id)
A02
```

Row or column coordinates A series of WellRecord objects contiguous to each other in the plate can be retrieved in bulk by using the python list slicing syntax on PlateRecord objects; rows and columns are numbered with a 0-based index.

```
>>> print(record[0])
Plate ID: PM09
Well: 12
Rows: 1
Columns: 12
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ..., WellRecord['A12']')
>>> print(record[:, 0])
Plate ID: PM09
Well: 8
Rows: 8
Columns: 1
PlateRecord('WellRecord['A01'], WellRecord['B01'], WellRecord['C01'], ..., WellRecord['H01']')
>>> print(record[:3, :3])
Plate ID: PM09
Well: 9
Rows: 3
Columns: 3
PlateRecord('WellRecord['A01'], WellRecord['A02'], WellRecord['A03'], ..., WellRecord['C03']')
```

21.1.2 Manipulating Phenotype Microarray data

21.1.2.1 Accessing raw data

The raw data extracted from the PM files is comprised of a series of tuples for each well, containing the time (in hours) and the colorimetric measure (in arbitrary units). Usually the instrument collects data every fifteen minutes, but that can vary between experiments. The raw data can be accessed by iterating on a WellRecord object; in the example below only the first ten time points are shown.

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> well = record["A02"]
```

```
>>> for time, signal in well:
...     print(time, signal)
...
(0.0, 12.0)
(0.25, 18.0)
(0.5, 27.0)
(0.75, 35.0)
(1.0, 37.0)
(1.25, 41.0)
(1.5, 44.0)
(1.75, 44.0)
(2.0, 44.0)
(2.25, 44.0)
[...]
```

This method, while providing a way to access the raw data, doesn't allow a direct comparison between different WellRecord objects, which may have measurements at different time points.

21.1.2.2 Accessing interpolated data

To make it easier to compare different experiments and in general to allow a more intuitive handling of the phenotypic data, the module allows to define a custom slicing of the time points that are present in the WellRecord object. Colorimetric data for time points that have not been directly measured are derived through a linear interpolation of the available data, otherwise a NaN is returned. This method only works in the time interval where actual data is available. Time intervals can be defined with the same syntax as list indexing; the default time interval is therefore one hour.

```
>>> well[:10]
[12.0, 37.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0]
```

Different time intervals can be used, for instance five minutes:

```
>>> well[63:64:0.083]
[12.0, 37.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0, 44.0]
>>> well[9.55]
44.0
>>> well[63.33:73.33]
[113.31999999999999,
 117.0,
 120.31999999999999,
 128.0,
 129.63999999999999,
 132.95999999999998,
 136.95999999999998,
 140.0,
 142.0,
 nan]
```

21.1.2.3 Control well subtraction

Many Phenotype Microarray plates contain a control well (usually A01), that is a well where the media shouldn't support any growth; the low signal produced by this well can be subtracted from the other wells. The PlateRecord objects have a dedicated function for that, which returns another PlateRecord object with the corrected data.

```
>>> corrected = record.subtract_control(control="A01")
>>> record["A01"][63]
336.0
>>> corrected["A01"][63]
0.0
```

21.1.2.4 Parameters extraction

Those wells where metabolic activity is observed show a sigmoid behavior for the colorimetric data. To allow an easier way to compare different experiments a sigmoid curve can be fitted onto the data, so that a series of summary parameters can be extracted and used for comparisons. The parameters that can be extracted from the curve are:

- Minimum (**min**) and maximum (**max**) signal;
- Average height (**average_height**);
- Area under the curve (**area**);
- Curve plateau point (**plateau**);
- Curve slope during exponential metabolic activity (**slope**);
- Curve lag time (**lag**).

All the parameters (except **min**, **max** and **average_height**) require the scipy library to be installed. The fit function uses three sigmoid functions:

Gompertz $Ae^{-e^{(\frac{\mu_m e}{A}(\lambda-t)+1)}} + y0$

Logistic $\frac{A}{1+e^{(\frac{4\mu_m}{A}(\lambda-t)+2)}} + y_0$

Richards $A(1 + ve^{1+v} + e^{\frac{\mu_m}{A}(1+v)(1+\frac{1}{v})(\lambda-t)})^{-\frac{1}{v}} + y0$

Where:

A corresponds to the **plateau**

μ_m corresponds to the **slope**

λ corresponds to the **lag**

These functions have been derived from this publication. The fit method by default tries first to fit the gompertz function: if it fails it will then try to fit the logistic and then the richards function. The user can also specify one of the three functions to be applied.

```
>>> from Bio import phenotype
>>> record = list(phenotype.parse("Plates.csv", "pm-csv"))[-1]
>>> well = record["A02"]
>>> well.fit()
>>> print("Function fitted: %s" % well.model)
Function fitted: gompertz
>>> for param in ["area", "average_height", "lag", "max", "min", "plateau", "slope"]:
...     print("%s\t%.2f" % (param, getattr(well, param)))
...
area    4414.38
```

```
average_height  61.58
lag     48.60
max     143.00
min     12.00
plateau 120.02
slope   4.99
```

21.1.3 Writing Phenotype Microarray data

PlateRecord objects can be written to file in the form of JSON files, a format compatible with other software packages such as opm or DuctApe.

```
>>> phenotype.write(record, "out.json", "pm-json")
1
```

Chapter 22

Cookbook – Cool things to do with it

Biopython now has two collections of "cookbook" examples – this chapter (which has been included in this tutorial for many years and has gradually grown), and http://biopython.org/wiki/Category:Cookbook which is a user contributed collection on our wiki.

We're trying to encourage Biopython users to contribute their own examples to the wiki. In addition to helping the community, one direct benefit of sharing an example like this is that you could also get some feedback on the code from other Biopython users and developers - which could help you improve all your Python code.

In the long term, we may end up moving all of the examples in this chapter to the wiki, or elsewhere within the tutorial.

22.1 Working with sequence files

This section shows some more examples of sequence input/output, using the `Bio.SeqIO` module described in Chapter 5.

22.1.1 Filtering a sequence file

Often you'll have a large file with many sequences in it (e.g. FASTA file or genes, or a FASTQ or SFF file of reads), a separate shorter list of the IDs for a subset of sequences of interest, and want to make a new sequence file for this subset.

Let's say the list of IDs is in a simple text file, as the first word on each line. This could be a tabular file where the first column is the ID. Try something like this:

```
from Bio import SeqIO

input_file = "big_file.sff"
id_file = "short_list.txt"
output_file = "short_list.sff"

with open(id_file) as id_handle:
    wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
print("Found %i unique identifiers in %s" % (len(wanted), id_file))

records = (r for r in SeqIO.parse(input_file, "sff") if r.id in wanted)
count = SeqIO.write(records, output_file, "sff")
print("Saved %i records from %s to %s" % (count, input_file, output_file))
```

```
if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))
```

Note that we use a Python `set` rather than a `list`, this makes testing membership faster.

As discussed in Section 5.6, for a large FASTA or FASTQ file for speed you would be better off not using the high-level `SeqIO` interface, but working directly with strings. This next example shows how to do this with FASTQ files – it is more complicated:

```
from Bio.SeqIO.QualityIO import FastqGeneralIterator

input_file = "big_file.fastq"
id_file = "short_list.txt"
output_file = "short_list.fastq"

with open(id_file) as id_handle:
    # Taking first word on each line as an identifier
    wanted = set(line.rstrip("\n").split(None, 1)[0] for line in id_handle)
print("Found %i unique identifiers in %s" % (len(wanted), id_file))

with open(input_file) as in_handle:
    with open(output_file, "w") as out_handle:
        for title, seq, qual in FastqGeneralIterator(in_handle):
            # The ID is the first word in the title line (after the @ sign):
            if title.split(None, 1)[0] in wanted:
                # This produces a standard 4-line FASTQ entry:
                out_handle.write("@%s\n%s\n+\n%s\n" % (title, seq, qual))
                count += 1
print("Saved %i records from %s to %s" % (count, input_file, output_file))
if count < len(wanted):
    print("Warning %i IDs not found in %s" % (len(wanted) - count, input_file))
```

22.1.2 Producing randomized genomes

Let's suppose you are looking at genome sequence, hunting for some sequence feature – maybe extreme local GC% bias, or possible restriction digest sites. Once you've got your Python code working on the real genome it may be sensible to try running the same search on randomized versions of the same genome for statistical analysis (after all, any "features" you've found could just be there just by chance).

For this discussion, we'll use the GenBank file for the pPCP1 plasmid from *Yersinia pestis biovar Microtus*. The file is included with the Biopython unit tests under the GenBank folder, or you can get it from our website, NC_005816.gb. This file contains one and only one record, so we can read it in as a `SeqRecord` using the `Bio.SeqIO.read()` function:

```
>>> from Bio import SeqIO
>>> original_rec = SeqIO.read("NC_005816.gb", "genbank")
```

So, how can we generate a shuffled versions of the original sequence? I would use the built in Python `random` module for this, in particular the function `random.shuffle` – but this works on a Python list. Our sequence is a `Seq` object, so in order to shuffle it we need to turn it into a list:

```
>>> import random
>>> nuc_list = list(original_rec.seq)
>>> random.shuffle(nuc_list)  # acts in situ!
```

Now, in order to use `Bio.SeqIO` to output the shuffled sequence, we need to construct a new `SeqRecord` with a new `Seq` object using this shuffled list. In order to do this, we need to turn the list of nucleotides (single letter strings) into a long string – the standard Python way to do this is with the string object's join method.

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> shuffled_rec = SeqRecord(
...     Seq("".join(nuc_list)), id="Shuffled", description="Based on %s" % original_rec.id
... )
```

Let's put all these pieces together to make a complete Python script which generates a single FASTA file containing 30 randomly shuffled versions of the original sequence.

This first version just uses a big for loop and writes out the records one by one (using the `SeqRecord`'s format method described in Section 5.5.4):

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

original_rec = SeqIO.read("NC_005816.gb", "genbank")

with open("shuffled.fasta", "w") as output_handle:
    for i in range(30):
        nuc_list = list(original_rec.seq)
        random.shuffle(nuc_list)
        shuffled_rec = SeqRecord(
            Seq("".join(nuc_list)),
            id="Shuffled%i" % (i + 1),
            description="Based on %s" % original_rec.id,
        )
        output_handle.write(shuffled_rec.format("fasta"))
```

Personally I prefer the following version using a function to shuffle the record and a generator expression instead of the for loop:

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO


def make_shuffle_record(record, new_id):
    nuc_list = list(record.seq)
    random.shuffle(nuc_list)
    return SeqRecord(
        Seq("".join(nuc_list)),
        id=new_id,
        description="Based on %s" % original_rec.id,
    )
```

```
original_rec = SeqIO.read("NC_005816.gb", "genbank")
shuffled_recs = (
    make_shuffle_record(original_rec, "Shuffled%i" % (i + 1)) for i in range(30)
)

SeqIO.write(shuffled_recs, "shuffled.fasta", "fasta")
```

22.1.3 Translating a FASTA file of CDS entries

Suppose you've got an input file of CDS entries for some organism, and you want to generate a new FASTA file containing their protein sequences. i.e. Take each nucleotide sequence from the original file, and translate it. Back in Section 3.8 we saw how to use the `Seq` object's `translate method`, and the optional `cds` argument which enables correct translation of alternative start codons.

We can combine this with `Bio.SeqIO` as shown in the reverse complement example in Section 5.5.3. The key point is that for each nucleotide `SeqRecord`, we need to create a protein `SeqRecord` - and take care of naming it.

You can write you own function to do this, choosing suitable protein identifiers for your sequences, and the appropriate genetic code. In this example we just use the default table and add a prefix to the identifier:

```
from Bio.SeqRecord import SeqRecord


def make_protein_record(nuc_record):
    """Returns a new SeqRecord with the translated sequence (default table)."""
    return SeqRecord(
        seq=nuc_record.seq.translate(cds=True),
        id="trans_" + nuc_record.id,
        description="translation of CDS, using default table",
    )
```

We can then use this function to turn the input nucleotide records into protein records ready for output. An elegant way and memory efficient way to do this is with a generator expression:

```
from Bio import SeqIO

proteins = (
    make_protein_record(nuc_rec)
    for nuc_rec in SeqIO.parse("coding_sequences.fasta", "fasta")
)
SeqIO.write(proteins, "translations.fasta", "fasta")
```

This should work on any FASTA file of complete coding sequences. If you are working on partial coding sequences, you may prefer to use `nuc_record.seq.translate(to_stop=True)` in the example above, as this wouldn't check for a valid start codon etc.

22.1.4 Making the sequences in a FASTA file upper case

Often you'll get data from collaborators as FASTA files, and sometimes the sequences can be in a mixture of upper and lower case. In some cases this is deliberate (e.g. lower case for poor quality regions), but usually it is not important. You may want to edit the file to make everything consistent (e.g. all upper case), and you can do this easily using the `upper()` method of the `SeqRecord` object (added in Biopython 1.55):

```
from Bio import SeqIO

records = (rec.upper() for rec in SeqIO.parse("mixed.fas", "fasta"))
count = SeqIO.write(records, "upper.fas", "fasta")
print("Converted %i records to upper case" % count)
```

How does this work? The first line is just importing the `Bio.SeqIO` module. The second line is the interesting bit – this is a Python generator expression which gives an upper case version of each record parsed from the input file (`mixed.fas`). In the third line we give this generator expression to the `Bio.SeqIO.write()` function and it saves the new upper cases records to our output file (`upper.fas`).

The reason we use a generator expression (rather than a list or list comprehension) is this means only one record is kept in memory at a time. This can be really important if you are dealing with large files with millions of entries.

22.1.5 Sorting a sequence file

Suppose you wanted to sort a sequence file by length (e.g. a set of contigs from an assembly), and you are working with a file format like FASTA or FASTQ which `Bio.SeqIO` can read, write (and index).

If the file is small enough, you can load it all into memory at once as a list of `SeqRecord` objects, sort the list, and save it:

```
from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

The only clever bit is specifying a comparison method for how to sort the records (here we sort them by length). If you wanted the longest records first, you could flip the comparison or use the reverse argument:

```
from Bio import SeqIO

records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(key=lambda r: -len(r))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

Now that's pretty straight forward - but what happens if you have a very large file and you can't load it all into memory like this? For example, you might have some next-generation sequencing reads to sort by length. This can be solved using the `Bio.SeqIO.index()` function.

```
from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted(
    (len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory
record_index = SeqIO.index("ls_orchid.fasta", "fasta")
records = (record_index[id] for id in ids)
SeqIO.write(records, "sorted.fasta", "fasta")
```

First we scan through the file once using `Bio.SeqIO.parse()`, recording the record identifiers and their lengths in a list of tuples. We then sort this list to get them in length order, and discard the lengths. Using this sorted list of identifiers `Bio.SeqIO.index()` allows us to retrieve the records one by one, and we pass them to `Bio.SeqIO.write()` for output.

These examples all use `Bio.SeqIO` to parse the records into `SeqRecord` objects which are output using `Bio.SeqIO.write()`. What if you want to sort a file format which `Bio.SeqIO.write()` doesn't support, like the plain text SwissProt format? Here is an alternative solution using the `get_raw()` method added to `Bio.SeqIO.index()` in Biopython 1.54 (see Section 5.4.2.2).

```
from Bio import SeqIO

# Get the lengths and ids, and sort on length
len_and_ids = sorted(
    (len(rec), rec.id) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

Note with Python 3 onwards, we have to open the file for writing in binary mode because the `get_raw()` method returns `bytes` objects.

As a bonus, because it doesn't parse the data into `SeqRecord` objects a second time it should be faster. If you only want to use this with FASTA format, we can speed this up one step further by using the low-level FASTA parser to get the record identifiers and lengths:

```
from Bio.SeqIO.FastaIO import SimpleFastaParser
from Bio import SeqIO

# Get the lengths and ids, and sort on length
with open("ls_orchid.fasta") as in_handle:
    len_and_ids = sorted(
        (len(seq), title.split(None, 1)[0])
        for title, seq in SimpleFastaParser(in_handle)
    )
ids = reversed([id for (length, id) in len_and_ids])
del len_and_ids  # free this memory

record_index = SeqIO.index("ls_orchid.fasta", "fasta")
with open("sorted.fasta", "wb") as out_handle:
    for id in ids:
        out_handle.write(record_index.get_raw(id))
```

22.1.6 Simple quality filtering for FASTQ files

The FASTQ file format was introduced at Sanger and is now widely used for holding nucleotide sequencing reads together with their quality scores. FASTQ files (and the related QUAL files) are an excellent example of per-letter-annotation, because for each nucleotide in the sequence there is an associated quality score. Any per-letter-annotation is held in a `SeqRecord` in the `letter_annotations` dictionary as a list, tuple or string (with the same number of elements as the sequence length).

One common task is taking a large set of sequencing reads and filtering them (or cropping them) based on their quality scores. The following example is very simplistic, but should illustrate the basics of working with quality data in a `SeqRecord` object. All we are going to do here is read in a file of FASTQ data, and filter it to pick out only those records whose PHRED quality scores are all above some threshold (here 20).

For this example we'll use some real data downloaded from the ENA sequence read archive, ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz (2MB) which unzips to a 19MB file `SRR020192.fastq`. This is some Roche 454 GS FLX single end data from virus infected California sea lions (see https://www.ebi.ac.uk/ena/data/view/SRS004476 for details).

First, let's count the reads:

```
from Bio import SeqIO

count = 0
for rec in SeqIO.parse("SRR020192.fastq", "fastq"):
    count += 1
print("%i reads" % count)
```

Now let's do a simple filtering for a minimum PHRED quality of 20:

```
from Bio import SeqIO

good_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if min(rec.letter_annotations["phred_quality"]) >= 20
)
count = SeqIO.write(good_reads, "good_quality.fastq", "fastq")
print("Saved %i reads" % count)
```

This pulled out only 14580 reads out of the 41892 present. A more sensible thing to do would be to quality trim the reads, but this is intended as an example only.

FASTQ files can contain millions of entries, so it is best to avoid loading them all into memory at once. This example uses a generator expression, which means only one `SeqRecord` is created at a time - avoiding any memory limitations.

Note that it would be faster to use the low-level `FastqGeneralIterator` parser here (see Section 5.6), but that does not turn the quality string into integer scores.

22.1.7 Trimming off primer sequences

For this example we're going to pretend that `GATGACGGTGT` is a 5' primer sequence we want to look for in some FASTQ formatted read data. As in the example above, we'll use the `SRR020192.fastq` file downloaded from the ENA (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz).

By using the main `Bio.SeqIO` interface, the same approach would work with any other supported file format (e.g. FASTA files). However, for large FASTQ files it would be faster the low-level `FastqGeneralIterator` parser here (see the earlier example, and Section 5.6).

This code uses `Bio.SeqIO` with a generator expression (to avoid loading all the sequences into memory at once), and the `Seq` object's `startswith` method to see if the read starts with the primer sequence:

```
from Bio import SeqIO

primer_reads = (
    rec
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
```

```
    if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(primer_reads, "with_primer.fastq", "fastq")
print("Saved %i reads" % count)
```

That should find 13819 reads from `SRR014849.fastq` and save them to a new FASTQ file, `with_primer.fastq`.

Now suppose that instead you wanted to make a FASTQ file containing these reads but with the primer sequence removed? That's just a small change as we can slice the `SeqRecord` (see Section 4.7) to remove the first eleven letters (the length of our primer):

```
from Bio import SeqIO

trimmed_primer_reads = (
    rec[11:]
    for rec in SeqIO.parse("SRR020192.fastq", "fastq")
    if rec.seq.startswith("GATGACGGTGT")
)
count = SeqIO.write(trimmed_primer_reads, "with_primer_trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Again, that should pull out the 13819 reads from `SRR020192.fastq`, but this time strip off the first ten characters, and save them to another new FASTQ file, `with_primer_trimmed.fastq`.

Now, suppose you want to create a new FASTQ file where these reads have their primer removed, but all the other reads are kept as they were? If we want to still use a generator expression, it is probably clearest to define our own trim function:

```
from Bio import SeqIO


def trim_primer(record, primer):
    if record.seq.startswith(primer):
        return record[len(primer) :]
    else:
        return record


trimmed_reads = (
    trim_primer(record, "GATGACGGTGT")
    for record in SeqIO.parse("SRR020192.fastq", "fastq")
)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This takes longer, as this time the output file contains all 41892 reads. Again, we're used a generator expression to avoid any memory problems. You could alternatively use a generator function rather than a generator expression.

```
from Bio import SeqIO


def trim_primers(records, primer):
    """Removes perfect primer sequences at start of reads.
```

```
    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
    len_primer = len(primer)  # cache this for later
    for record in records:
        if record.seq.startswith(primer):
            yield record[len_primer:]
        else:
            yield record


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_primers(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

This form is more flexible if you want to do something more complicated where only some of the records are retained – as shown in the next example.

22.1.8 Trimming off adaptor sequences

This is essentially a simple extension to the previous example. We are going to going to pretend `GATGACGGTGT` is an adaptor sequence in some FASTQ formatted read data, again the `SRR020192.fastq` file from the NCBI (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz).

This time however, we will look for the sequence *anywhere* in the reads, not just at the very beginning:

```
from Bio import SeqIO


def trim_adaptors(records, adaptor):
    """Trims perfect adaptor sequences.

    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        else:
            # trim off the adaptor
            yield record[index + len_adaptor :]


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT")
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

Because we are using a FASTQ input file in this example, the `SeqRecord` objects have per-letter-annotation for the quality scores. By slicing the `SeqRecord` object the appropriate scores are used on

the trimmed records, so we can output them as a FASTQ file too.

Compared to the output of the previous example where we only looked for a primer/adaptor at the start of each read, you may find some of the trimmed reads are quite short after trimming (e.g. if the adaptor was found in the middle rather than near the start). So, let's add a minimum length requirement as well:

```
from Bio import SeqIO


def trim_adaptors(records, adaptor, min_len):
    """Trims perfect adaptor sequences, checks read length.

    This is a generator function, the records argument should
    be a list or iterator returning SeqRecord objects.
    """
    len_adaptor = len(adaptor)  # cache this for later
    for record in records:
        len_record = len(record)  # cache this for later
        if len(record) < min_len:
            # Too short to keep
            continue
        index = record.seq.find(adaptor)
        if index == -1:
            # adaptor not found, so won't trim
            yield record
        elif len_record - index - len_adaptor >= min_len:
            # after trimming this will still be long enough
            yield record[index + len_adaptor :]


original_reads = SeqIO.parse("SRR020192.fastq", "fastq")
trimmed_reads = trim_adaptors(original_reads, "GATGACGGTGT", 100)
count = SeqIO.write(trimmed_reads, "trimmed.fastq", "fastq")
print("Saved %i reads" % count)
```

By changing the format names, you could apply this to FASTA files instead. This code also could be extended to do a fuzzy match instead of an exact match (maybe using a pairwise alignment, or taking into account the read quality scores), but that will be much slower.

22.1.9 Converting FASTQ files

Back in Section 5.5.2 we showed how to use `Bio.SeqIO` to convert between two file formats. Here we'll go into a little more detail regarding FASTQ files which are used in second generation DNA sequencing. Please refer to Cock *et al.* (2009) [6] for a longer description. FASTQ files store both the DNA sequence (as a string) and the associated read qualities.

PHRED scores (used in most FASTQ files, and also in QUAL files, ACE files and SFF files) have become a *de facto* standard for representing the probability of a sequencing error (here denoted by P_e) at a given base using a simple base ten log transformation:

$$Q_{\text{PHRED}} = -10 \times \log_{10}(P_e) \tag{22.1}$$

This means a wrong read ($P_e = 1$) gets a PHRED quality of 0, while a very good read like $P_e = 0.00001$ gets a PHRED quality of 50. While for raw sequencing data qualities higher than this are rare, with post

processing such as read mapping or assembly, qualities of up to about 90 are possible (indeed, the MAQ tool allows for PHRED scores in the range 0 to 93 inclusive).

The FASTQ format has the potential to become a *de facto* standard for storing the letters and quality scores for a sequencing read in a single plain text file. The only fly in the ointment is that there are at least three versions of the FASTQ format which are incompatible and difficult to distinguish...

1. The original Sanger FASTQ format uses PHRED qualities encoded with an ASCII offset of 33. The NCBI are using this format in their Short Read Archive. We call this the `fastq` (or `fastq-sanger`) format in `Bio.SeqIO`.

2. Solexa (later bought by Illumina) introduced their own version using Solexa qualities encoded with an ASCII offset of 64. We call this the `fastq-solexa` format.

3. Illumina pipeline 1.3 onwards produces FASTQ files with PHRED qualities (which is more consistent), but encoded with an ASCII offset of 64. We call this the `fastq-illumina` format.

The Solexa quality scores are defined using a different log transformation:

$$Q_{\text{Solexa}} = -10 \times \log_{10}\left(\frac{P_e}{1-P_e}\right) \tag{22.2}$$

Given Solexa/Illumina have now moved to using PHRED scores in version 1.3 of their pipeline, the Solexa quality scores will gradually fall out of use. If you equate the error estimates (P_e) these two equations allow conversion between the two scoring systems - and Biopython includes functions to do this in the `Bio.SeqIO.QualityIO` module, which are called if you use `Bio.SeqIO` to convert an old Solexa/Illumina file into a standard Sanger FASTQ file:

```
from Bio import SeqIO

SeqIO.convert("solexa.fastq", "fastq-solexa", "standard.fastq", "fastq")
```

If you want to convert a new Illumina 1.3+ FASTQ file, all that gets changed is the ASCII offset because although encoded differently the scores are all PHRED qualities:

```
from Bio import SeqIO

SeqIO.convert("illumina.fastq", "fastq-illumina", "standard.fastq", "fastq")
```

Note that using `Bio.SeqIO.convert()` like this is *much* faster than combining `Bio.SeqIO.parse()` and `Bio.SeqIO.write()` because optimized code is used for converting between FASTQ variants (and also for FASTQ to FASTA conversion).

For good quality reads, PHRED and Solexa scores are approximately equal, which means since both the `fasta-solexa` and `fastq-illumina` formats use an ASCII offset of 64 the files are almost the same. This was a deliberate design choice by Illumina, meaning applications expecting the old `fasta-solexa` style files will probably be OK using the newer `fastq-illumina` files (on good data). Of course, both variants are very different from the original FASTQ standard as used by Sanger, the NCBI, and elsewhere (format name `fastq` or `fastq-sanger`).

For more details, see the built in help (also online):

```
>>> from Bio.SeqIO import QualityIO
>>> help(QualityIO)
```

22.1.10 Converting FASTA and QUAL files into FASTQ files

FASTQ files hold *both* sequences and their quality strings. FASTA files hold *just* sequences, while QUAL files hold *just* the qualities. Therefore a single FASTQ file can be converted to or from *paired* FASTA and QUAL files.

Going from FASTQ to FASTA is easy:

```
from Bio import SeqIO

SeqIO.convert("example.fastq", "fastq", "example.fasta", "fasta")
```

Going from FASTQ to QUAL is also easy:

```
from Bio import SeqIO

SeqIO.convert("example.fastq", "fastq", "example.qual", "qual")
```

However, the reverse is a little more tricky. You can use `Bio.SeqIO.parse()` to iterate over the records in a *single* file, but in this case we have two input files. There are several strategies possible, but assuming that the two files are really paired the most memory efficient way is to loop over both together. The code is a little fiddly, so we provide a function called `PairedFastaQualIterator` in the `Bio.SeqIO.QualityIO` module to do this. This takes two handles (the FASTA file and the QUAL file) and returns a `SeqRecord` iterator:

```
from Bio.SeqIO.QualityIO import PairedFastaQualIterator

for record in PairedFastaQualIterator(open("example.fasta"), open("example.qual")):
    print(record)
```

This function will check that the FASTA and QUAL files are consistent (e.g. the records are in the same order, and have the same sequence length). You can combine this with the `Bio.SeqIO.write()` function to convert a pair of FASTA and QUAL files into a single FASTQ files:

```
from Bio import SeqIO
from Bio.SeqIO.QualityIO import PairedFastaQualIterator

with open("example.fasta") as f_handle, open("example.qual") as q_handle:
    records = PairedFastaQualIterator(f_handle, q_handle)
    count = SeqIO.write(records, "temp.fastq", "fastq")
print("Converted %i records" % count)
```

22.1.11 Indexing a FASTQ file

FASTQ files are usually very large, with millions of reads in them. Due to the sheer amount of data, you can't load all the records into memory at once. This is why the examples above (filtering and trimming) iterate over the file looking at just one `SeqRecord` at a time.

However, sometimes you can't use a big loop or an iterator - you may need random access to the reads. Here the `Bio.SeqIO.index()` function may prove very helpful, as it allows you to access any read in the FASTQ file by its name (see Section 5.4.2).

Again we'll use the `SRR020192.fastq` file from the ENA (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz), although this is actually quite a small FASTQ file with less than $50,000$ reads:

```
>>> from Bio import SeqIO
>>> fq_dict = SeqIO.index("SRR020192.fastq", "fastq")
>>> len(fq_dict)
41892
>>> list(fq_dict.keys())[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
Seq('GTCCCAGTATTCGGATTTGTCTGCCAAAACAATGAAATTGACACAGTTTACAAC...CCG')
```

When testing this on a FASTQ file with seven million reads, indexing took about a minute, but record access was almost instant.

The sister function `Bio.SeqIO.index_db()` lets you save the index to an SQLite3 database file for near instantaneous reuse - see Section 5.4.2 for more details.

The example in Section 22.1.5 show how you can use the `Bio.SeqIO.index()` function to sort a large FASTA file – this could also be used on FASTQ files.

22.1.12 Converting SFF files

If you work with 454 (Roche) sequence data, you will probably have access to the raw data as a Standard Flowgram Format (SFF) file. This contains the sequence reads (called bases) with quality scores and the original flow information.

A common task is to convert from SFF to a pair of FASTA and QUAL files, or to a single FASTQ file. These operations are trivial using the `Bio.SeqIO.convert()` function (see Section 5.5.2):

```
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.qual", "qual")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff", "reads.fastq", "fastq")
10
```

Remember the convert function returns the number of records, in this example just ten. This will give you the *untrimmed* reads, where the leading and trailing poor quality sequence or adaptor will be in lower case. If you want the *trimmed* reads (using the clipping information recorded within the SFF file) use this:

```
>>> from Bio import SeqIO
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fasta", "fasta")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.qual", "qual")
10
>>> SeqIO.convert("E3MFGYR02_random_10_reads.sff", "sff-trim", "trimmed.fastq", "fastq")
10
```

If you run Linux, you could ask Roche for a copy of their "off instrument" tools (often referred to as the Newbler tools). This offers an alternative way to do SFF to FASTA or QUAL conversion at the command line (but currently FASTQ output is not supported), e.g.

```
$ sffinfo -seq -notrim E3MFGYR02_random_10_reads.sff > reads.fasta
$ sffinfo -qual -notrim E3MFGYR02_random_10_reads.sff > reads.qual
$ sffinfo -seq -trim E3MFGYR02_random_10_reads.sff > trimmed.fasta
$ sffinfo -qual -trim E3MFGYR02_random_10_reads.sff > trimmed.qual
```

The way Biopython uses mixed case sequence strings to represent the trimming points deliberately mimics what the Roche tools do.

For more information on the Biopython SFF support, consult the built in help:

```
>>> from Bio.SeqIO import SffIO
>>> help(SffIO)
```

22.1.13 Identifying open reading frames

A very simplistic first step at identifying possible genes is to look for open reading frames (ORFs). By this we mean look in all six frames for long regions without stop codons – an ORF is just a region of nucleotides with no in frame stop codons.

Of course, to find a gene you would also need to worry about locating a start codon, possible promoters – and in Eukaryotes there are introns to worry about too. However, this approach is still useful in viruses and Prokaryotes.

To show how you might approach this with Biopython, we'll need a sequence to search, and as an example we'll again use the bacterial plasmid – although this time we'll start with a plain FASTA file with no pre-marked genes: NC_005816.fna. This is a bacterial sequence, so we'll want to use NCBI codon table 11 (see Section 3.8 about translation).

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> table = 11
>>> min_pro_len = 100
```

Here is a neat trick using the `Seq` object's `split` method to get a list of all the possible ORF translations in the six reading frames:

```
>>> for strand, nuc in [(+1, record.seq), (-1, record.seq.reverse_complement())]:
...     for frame in range(3):
...         length = 3 * ((len(record) - frame) // 3)  # Multiple of three
...         for pro in nuc[frame : frame + length].translate(table).split("*"):
...             if len(pro) >= min_pro_len:
...                 print(
...                     "%s...%s - length %i, strand %i, frame %i"
...                     % (pro[:30], pro[-3:], len(pro), strand, frame)
...                 )
...
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, frame 0
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, frame 1
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, frame 1
VKKILYIKALFLCTVIKLRRFIFSVNNMKF...DLP - length 165, strand 1, frame 1
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, frame 2
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, frame 2
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, frame 2
QGSGYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, frame 0
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, frame 0
WGKLQVIGLSMWMVLFSQRFDDWLNEQEDA...ESK - length 125, strand -1, frame 1
RGIFMSDTMVVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, frame 1
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, frame 1
LSHTVTDFTDQMAQVGLCQCVNVFLDEVTG...KAA - length 107, strand -1, frame 2
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, frame 2
```

Note that here we are counting the frames from the 5' end (start) of *each* strand. It is sometimes easier to always count from the 5' end (start) of the *forward* strand.

You could easily edit the above loop based code to build up a list of the candidate proteins, or convert this to a list comprehension. Now, one thing this code doesn't do is keep track of where the proteins are.

You could tackle this in several ways. For example, the following code tracks the locations in terms of the protein counting, and converts back to the parent sequence by multiplying by three, then adjusting for the frame and strand:

```
from Bio import SeqIO

record = SeqIO.read("NC_005816.gb", "genbank")
table = 11
min_pro_len = 100


def find_orfs_with_trans(seq, trans_table, min_protein_length):
    answer = []
    seq_len = len(seq)
    for strand, nuc in [(+1, seq), (-1, seq.reverse_complement())]:
        for frame in range(3):
            trans = nuc[frame:].translate(trans_table)
            trans_len = len(trans)
            aa_start = 0
            aa_end = 0
            while aa_start < trans_len:
                aa_end = trans.find("*", aa_start)
                if aa_end == -1:
                    aa_end = trans_len
                if aa_end - aa_start >= min_protein_length:
                    if strand == 1:
                        start = frame + aa_start * 3
                        end = min(seq_len, frame + aa_end * 3 + 3)
                    else:
                        start = seq_len - frame - aa_end * 3 - 3
                        end = seq_len - frame - aa_start * 3
                    answer.append((start, end, strand, trans[aa_start:aa_end]))
                aa_start = aa_end + 1
    answer.sort()
    return answer


orf_list = find_orfs_with_trans(record.seq, table, min_pro_len)
for start, end, strand, pro in orf_list:
    print(
        "%s...%s - length %i, strand %i, %i:%i"
        % (pro[:30], pro[-3:], len(pro), strand, start, end)
    )
```

And the output:

```
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
```

```
KSGELRQTPPASSTLHLRLILQRSGVMMEL...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELRYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKKRRPQKRPRRRRFFHRLRPPDE...PTR - length 128, strand 1, 3470:3857
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, 4249:4780
RGIFMSDTMVVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, 4814:5900
VKKILYIKALFLCTVIKLRRFIFSVNNMKF...DLP - length 165, strand 1, 5923:6421
LSHTVTDFTDQMAQVGLCQCVNVFLDEVTG...KAA - length 107, strand -1, 5974:6298
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, 6654:7602
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, 7788:8124
WGKLQVIGLSMWMVLFSQRFDDWLNEQEDA...ESK - length 125, strand -1, 8087:8465
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, 8741:9044
QGSGYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, 9264:9609
```

If you comment out the sort statement, then the protein sequences will be shown in the same order as before, so you can check this is doing the same thing. Here we have sorted them by location to make it easier to compare to the actual annotation in the GenBank file (as visualized in Section 19.1.9).

If however all you want to find are the locations of the open reading frames, then it is a waste of time to translate every possible codon, including doing the reverse complement to search the reverse strand too. All you need to do is search for the possible stop codons (and their reverse complements). Using regular expressions is an obvious approach here (see the Python module `re`). These are an extremely powerful (but rather complex) way of describing search strings, which are supported in lots of programming languages and also command line tools like `grep` as well). You can find whole books about this topic!

22.2 Sequence parsing plus simple plots

This section shows some more examples of sequence parsing, using the `Bio.SeqIO` module described in Chapter 5, plus the Python library matplotlib's `pylab` plotting interface (see the matplotlib website for a tutorial). Note that to follow these examples you will need matplotlib installed - but without it you can still try the data parsing bits.

22.2.1 Histogram of sequence lengths

There are lots of times when you might want to visualize the distribution of sequence lengths in a dataset – for example the range of contig sizes in a genome assembly project. In this example we'll reuse our orchid FASTA file ls_orchid.fasta which has only 94 sequences.

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the sequence lengths. You could do this with a for loop, but I find a list comprehension more pleasing:

```
>>> from Bio import SeqIO
>>> sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
>>> len(sizes), min(sizes), max(sizes)
(94, 572, 789)
>>> sizes
[740, 753, 748, 744, 733, 718, 730, 704, 740, 709, 700, 726, ..., 592]
```

Now that we have the lengths of all the genes (as a list of integers), we can use the matplotlib histogram function to display it.

```
from Bio import SeqIO

sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
```

```
import pylab

pylab.hist(sizes, bins=20)
pylab.title(
    "%i orchid sequences\nLengths %i to %i" % (len(sizes), min(sizes), max(sizes))
)
pylab.xlabel("Sequence length (bp)")
pylab.ylabel("Count")
pylab.show()
```

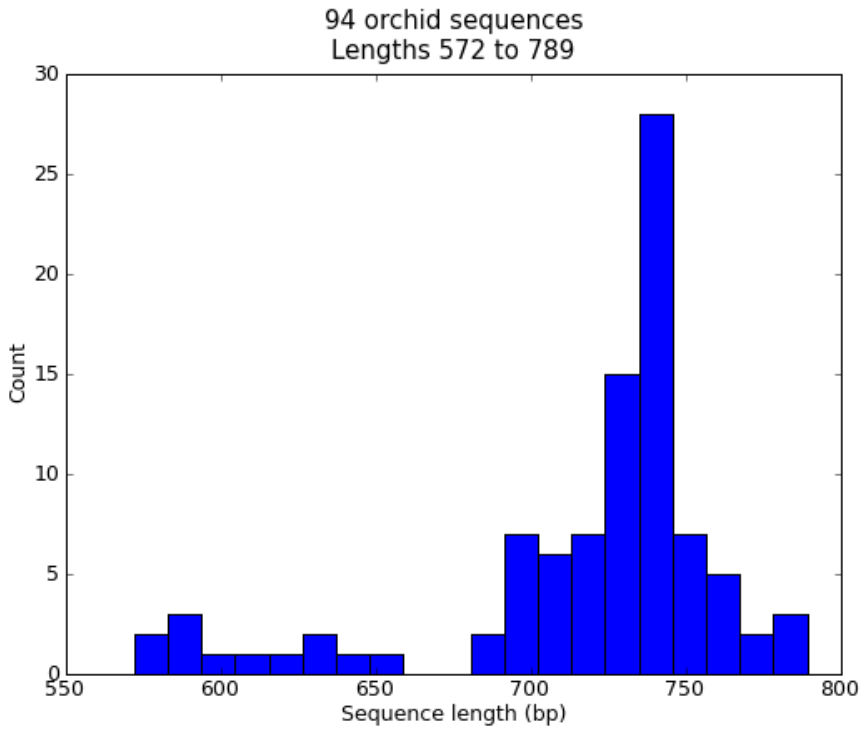


Figure 22.1: Histogram of orchid sequence lengths.

That should pop up a new window containing the graph shown in Figure 22.1. Notice that most of these orchid sequences are about 740 bp long, and there could be two distinct classes of sequence here with a subset of shorter sequences.

Tip: Rather than using `pylab.show()` to show the plot in a window, you can also use `pylab.savefig(...)` to save the figure to a file (e.g. as a PNG or PDF).

22.2.2 Plot of sequence GC%

Another easily calculated quantity of a nucleotide sequence is the GC%. You might want to look at the GC% of all the genes in a bacterial genome for example, and investigate any outliers which could have been recently acquired by horizontal gene transfer. Again, for this example we'll reuse our orchid FASTA file ls_orchid.fasta.

First of all, we will use `Bio.SeqIO` to parse the FASTA file and compile a list of all the GC percentages. Again, you could do this with a for loop, but I prefer this:

```
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

gc_values = sorted(
    100 * gc_fraction(rec.seq) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")
)
```

Having read in each sequence and calculated the GC%, we then sorted them into ascending order. Now we'll take this list of floating point values and plot them with matplotlib:

```
import pylab

pylab.plot(gc_values)
pylab.title(
    "%i orchid sequences\nGC%% %0.1f to %0.1f"
    % (len(gc_values), min(gc_values), max(gc_values))
)
pylab.xlabel("Genes")
pylab.ylabel("GC%")
pylab.show()
```

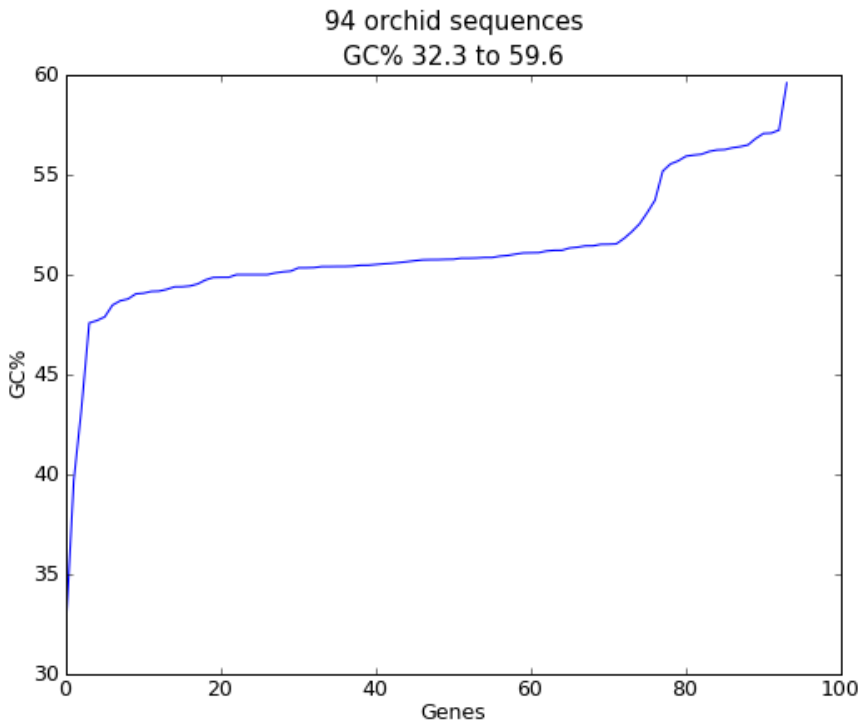


Figure 22.2: Histogram of orchid sequence lengths.

As in the previous example, that should pop up a new window with the graph shown in Figure 22.2. If you tried this on the full set of genes from one organism, you'd probably get a much smoother plot than this.

22.2.3 Nucleotide dot plots

A dot plot is a way of visually comparing two nucleotide sequences for similarity to each other. A sliding window is used to compare short sub-sequences to each other, often with a mismatch threshold. Here for simplicity we'll only look for perfect matches (shown in black in Figure 22.3).

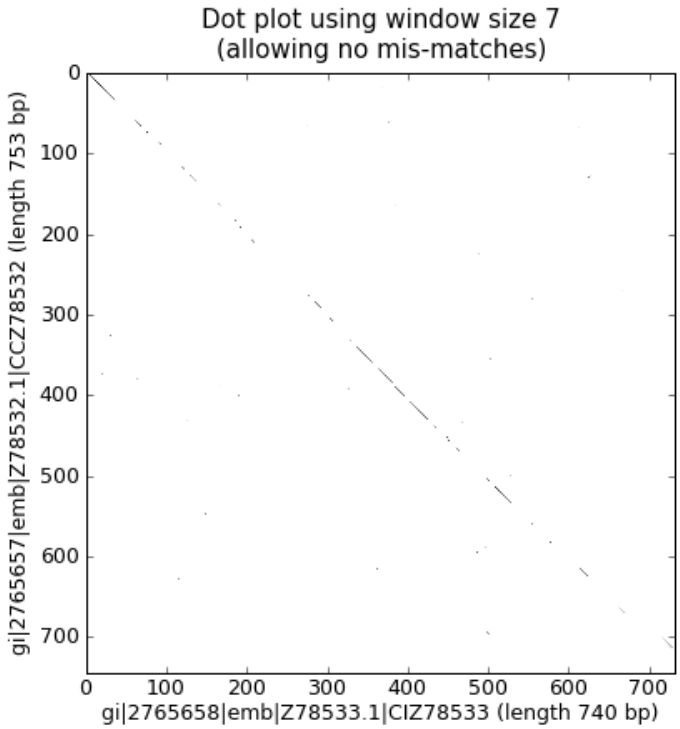


Figure 22.3: Nucleotide dot plot of two orchid sequence lengths (using pylab's imshow function).

To start off, we'll need two sequences. For the sake of argument, we'll just take the first two from our orchid FASTA file ls_orchid.fasta:

```
from Bio import SeqIO

with open("ls_orchid.fasta") as in_handle:
    record_iterator = SeqIO.parse(in_handle, "fasta")
    rec_one = next(record_iterator)
    rec_two = next(record_iterator)
```

We're going to show two approaches. Firstly, a simple naive implementation which compares all the window sized sub-sequences to each other to compiles a similarity matrix. You could construct a matrix or array object, but here we just use a list of lists of booleans created with a nested list comprehension:

```
window = 7
seq_one = rec_one.seq.upper()
seq_two = rec_two.seq.upper()
data = [
    [
        (seq_one[i : i + window] != seq_two[j : j + window])
```

```
        for j in range(len(seq_one) - window)
    ]
    for i in range(len(seq_two) - window)
]
```

Note that we have *not* checked for reverse complement matches here. Now we'll use the matplotlib's `pylab.imshow()` function to display this data, first requesting the gray color scheme so this is done in black and white:

```
import pylab

pylab.gray()
pylab.imshow(data)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Figure 22.3. As you might have expected, these two sequences are very similar with a partial line of window sized matches along the diagonal. There are no off diagonal matches which would be indicative of inversions or other interesting events.

The above code works fine on small examples, but there are two problems applying this to larger sequences, which we will address below. First off all, this brute force approach to the all against all comparisons is very slow. Instead, we'll compile dictionaries mapping the window sized sub-sequences to their locations, and then take the set intersection to find those sub-sequences found in both sequences. This uses more memory, but is *much* faster. Secondly, the `pylab.imshow()` function is limited in the size of matrix it can display. As an alternative, we'll use the `pylab.scatter()` function.

We start by creating dictionaries mapping the window-sized sub-sequences to locations:

```
window = 7
dict_one = {}
dict_two = {}
for seq, section_dict in [
    (rec_one.seq.upper(), dict_one),
    (rec_two.seq.upper(), dict_two),
]:
    for i in range(len(seq) - window):
        section = seq[i : i + window]
        try:
            section_dict[section].append(i)
        except KeyError:
            section_dict[section] = [i]
# Now find any sub-sequences found in both sequences
matches = set(dict_one).intersection(dict_two)
print("%i unique matches" % len(matches))
```

In order to use the `pylab.scatter()` we need separate lists for the x and y coordinates:

```
# Create lists of x and y coordinates for scatter plot
x = []
y = []
for section in matches:
    for i in dict_one[section]:
```

```
        for j in dict_two[section]:
            x.append(i)
            y.append(j)
```

We are now ready to draw the revised dot plot as a scatter plot:

```
import pylab

pylab.cla()  # clear any prior graph
pylab.gray()
pylab.scatter(x, y)
pylab.xlim(0, len(rec_one) - window)
pylab.ylim(0, len(rec_two) - window)
pylab.xlabel("%s (length %i bp)" % (rec_one.id, len(rec_one)))
pylab.ylabel("%s (length %i bp)" % (rec_two.id, len(rec_two)))
pylab.title("Dot plot using window size %i\n(allowing no mis-matches)" % window)
pylab.show()
```

That should pop up a new window showing the graph in Figure 22.4. Personally I find this second plot

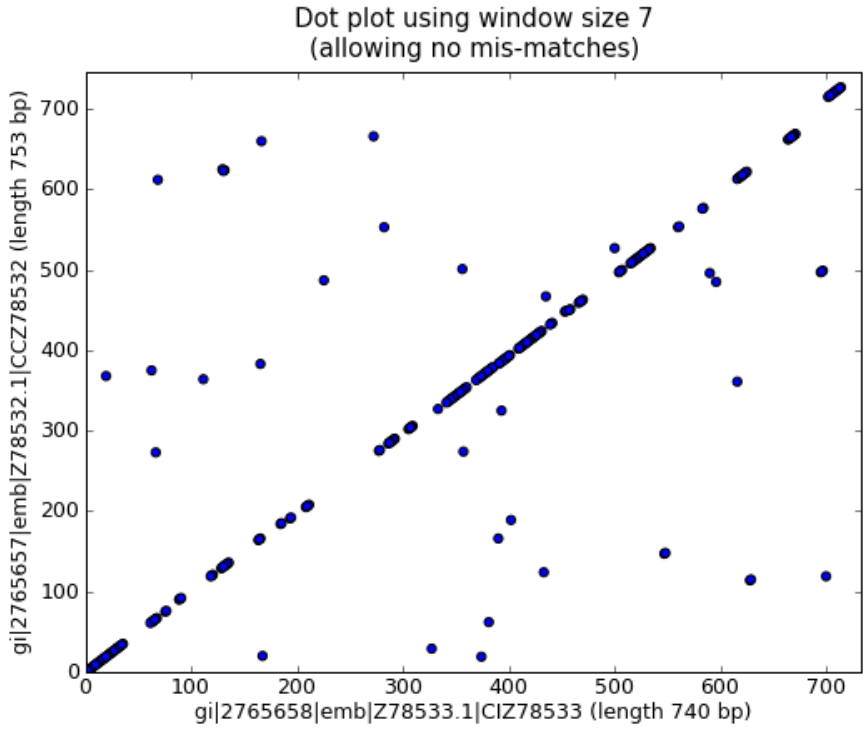


Figure 22.4: Nucleotide dot plot of two orchid sequence lengths (using pylab's scatter function).

much easier to read! Again note that we have *not* checked for reverse complement matches here – you could extend this example to do this, and perhaps plot the forward matches in one color and the reverse matches in another.

22.2.4 Plotting the quality scores of sequencing read data

If you are working with second generation sequencing data, you may want to try plotting the quality data. Here is an example using two FASTQ files containing paired end reads, `SRR001666_1.fastq` for the forward reads, and `SRR001666_2.fastq` for the reverse reads. These were downloaded from the ENA sequence read archive FTP site (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_1.fastq.gz and ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_2.fastq.gz), and are from *E. coli* – see https://www.ebi.ac.uk/ena/data/view/SRR001666 for details.

In the following code the `pylab.subplot(...)` function is used in order to show the forward and reverse qualities on two subplots, side by side. There is also a little bit of code to only plot the first fifty reads.

```
import pylab
from Bio import SeqIO

for subfigure in [1, 2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i, record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50:
            break  # trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.ylim(0, 45)
    pylab.ylabel("PHRED quality score")
    pylab.xlabel("Position")
pylab.savefig("SRR001666.png")
print("Done")
```

You should note that we are using the `Bio.SeqIO` format name `fastq` here because the NCBI has saved these reads using the standard Sanger FASTQ format with PHRED scores. However, as you might guess from the read lengths, this data was from an Illumina Genome Analyzer and was probably originally in one of the two Solexa/Illumina FASTQ variant file formats instead.

This example uses the `pylab.savefig(...)` function instead of `pylab.show(...)`, but as mentioned before both are useful. The result is shown in Figure 22.5.

22.3 BioSQL – storing sequences in a relational database

BioSQL is a joint effort between the OBF projects (BioPerl, BioJava etc) to support a shared database schema for storing sequence data. In theory, you could load a GenBank file into the database with BioPerl, then using Biopython extract this from the database as a record object with features - and get more or less the same thing as if you had loaded the GenBank file directly as a SeqRecord using `Bio.SeqIO` (Chapter 5).

Biopython's BioSQL module is currently documented at http://biopython.org/wiki/BioSQL which is part of our wiki pages.

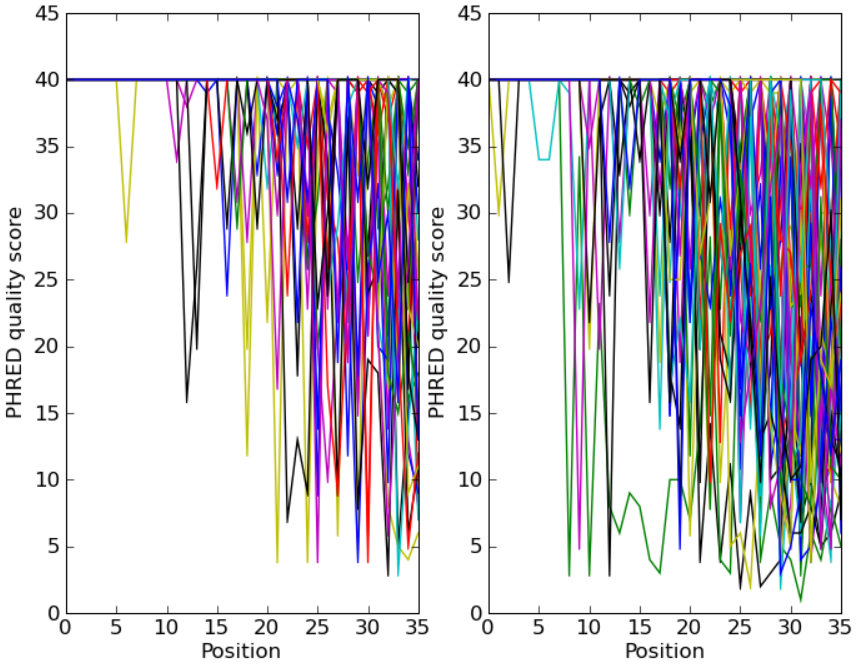


Figure 22.5: Quality plot for some paired end reads.

Chapter 23

The Biopython testing framework

Biopython has a regression testing framework (the file `run_tests.py`) based on unittest, the standard unit testing framework for Python. Providing comprehensive tests for modules is one of the most important aspects of making sure that the Biopython code is as bug-free as possible before going out. It also tends to be one of the most undervalued aspects of contributing. This chapter is designed to make running the Biopython tests and writing good test code as easy as possible. Ideally, every module that goes into Biopython should have a test (and should also have documentation!). All our developers, and anyone installing Biopython from source, are strongly encouraged to run the unit tests.

23.1 Running the tests

When you download the Biopython source code, or check it out from our source code repository, you should find a subdirectory call `Tests`. This contains the key script `run_tests.py`, lots of individual scripts named `test_XXX.py`, and lots of other subdirectories which contain input files for the test suite.

As part of building and installing Biopython you will typically run the full test suite at the command line from the Biopython source top level directory using the following:

```
$ python setup.py test
```

This is actually equivalent to going to the `Tests` subdirectory and running:

```
$ python run_tests.py
```

You'll often want to run just some of the tests, and this is done like this:

```
$ python run_tests.py test_SeqIO.py test_AlignIO.py
```

When giving the list of tests, the `.py` extension is optional, so you can also just type:

```
$ python run_tests.py test_SeqIO test_AlignIO
```

To run the docstring tests (see section 23.3), you can use

```
$ python run_tests.py doctest
```

You can also skip any tests which have been setup with an explicit online component by adding `--offline`, e.g.

```
$ python run_tests.py --offline
```

By default, `run_tests.py` runs all tests, including the docstring tests.

If an individual test is failing, you can also try running it directly, which may give you more information.

Tests based on Python's standard `unittest` framework will `import unittest` and then define `unittest.TestCase` classes, each with one or more sub-tests as methods starting with `test_` which check some specific aspect of the code.

23.1.1 Running the tests using Tox

Like most Python projects, you can also use Tox to run the tests on multiple Python versions, provided they are already installed in your system.

We do not provide the configuration `tox.ini` file in our code base because of difficulties pinning down user-specific settings (e.g. executable names of the Python versions). You may also only be interested in testing Biopython only against a subset of the Python versions that we support.

If you are interested in using Tox, you could start with the example `tox.ini` shown below:

```
[tox]
envlist = pypy,py38,py39

[testenv]
changedir = Tests
commands = {envpython} run_tests.py --offline
deps =
    numpy
    reportlab
```

Using the template above, executing `tox` will test your Biopython code against PyPy, Python 3.8 and 3.9. It assumes that those Pythons' executables are named "python3.8" for Python 3.8, and so on.

23.2 Writing tests

Let's say you want to write some tests for a module called `Biospam`. This can be a module you wrote, or an existing module that doesn't have any tests yet. In the examples below, we assume that `Biospam` is a module that does simple math.

Each Biopython test consists of a script containing the test itself, and optionally a directory with input files used by the test:

1. `test_Biospam.py` – The actual test code for your module.
2. `Biospam` [optional]– A directory where any necessary input files will be located. If you have any output files that should be manually reviewed, output them here (but this is discouraged) to prevent clogging up the main Tests directory. In general, use a temporary file/folder.

Any script with a `test_` prefix in the `Tests` directory will be found and run by `run_tests.py`. Below, we show an example test script `test_Biospam.py`. If you put this script in the Biopython `Tests` directory, then `run_tests.py` will find it and execute the tests contained in it:

```
$ python run_tests.py
test_Ace ... ok
test_AlignIO ... ok
test_BioSQL ... ok
test_BioSQL_SeqIO ... ok
test_Biospam ... ok
```

```
test_CAPS ... ok
test_Clustalw ... ok
...
----------------------------------------------------------------------
Ran 107 tests in 86.127 seconds
```

23.2.1 Writing a test using `unittest`

The `unittest`-framework has been included with Python since version 2.1, and is documented in the Python Library Reference (which I know you are keeping under your pillow, as recommended). There is also online documentation for unittest. If you are familiar with the `unittest` system (or something similar like the nose test framework), you shouldn't have any trouble. You may find looking at the existing examples within Biopython helpful too.

Here's a minimal `unittest`-style test script for `Biospam`, which you can copy and paste to get started:

```
import unittest
from Bio import Biospam


class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)


class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        result = Biospam.division(10.0, -2.0)
        self.assertAlmostEqual(result, -5.0)


if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

In the division tests, we use `assertAlmostEqual` instead of `assertEqual` to avoid tests failing due to roundoff errors; see the `unittest` chapter in the Python documentation for details and for other functionality available in `unittest` (online reference).

These are the key points of `unittest`-based tests:

- Test cases are stored in classes that derive from `unittest.TestCase` and cover one basic aspect of your code
- You can use methods `setUp` and `tearDown` for any repeated code which should be run before and after each test method. For example, the `setUp` method might be used to create an instance of the object

you are testing, or open a file handle. The `tearDown` should do any "tidying up", for example closing the file handle.

- The tests are prefixed with `test_` and each test should cover one specific part of what you are trying to test. You can have as many tests as you want in a class.
- At the end of the test script, you can use

```
if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

to execute the tests when the script is run by itself (rather than imported from `run_tests.py`). If you run this script, then you'll see something like the following:

```
$ python test_BiospamMyModule.py
test_addition1 (__main__.TestAddition) ... ok
test_addition2 (__main__.TestAddition) ... ok
test_division1 (__main__.TestDivision) ... ok
test_division2 (__main__.TestDivision) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.059s

OK
```

- To indicate more clearly what each test is doing, you can add docstrings to each test. These are shown when running the tests, which can be useful information if a test is failing.

```
import unittest
from Bio import Biospam


class BiospamTestAddition(unittest.TestCase):
    def test_addition1(self):
        """An addition test"""
        result = Biospam.addition(2, 3)
        self.assertEqual(result, 5)

    def test_addition2(self):
        """A second addition test"""
        result = Biospam.addition(9, -1)
        self.assertEqual(result, 8)


class BiospamTestDivision(unittest.TestCase):
    def test_division1(self):
        """Now let's check division"""
        result = Biospam.division(3.0, 2.0)
        self.assertAlmostEqual(result, 1.5)

    def test_division2(self):
        """A second division test"""
        result = Biospam.division(10.0, -2.0)
```

```
        self.assertAlmostEqual(result, -5.0)


if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity=2)
    unittest.main(testRunner=runner)
```

Running the script will now show you:

```
$ python test_BiospamMyModule.py
An addition test ... ok
A second addition test ... ok
Now let's check division ... ok
A second division test ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

If your module contains docstring tests (see section 23.3), you *may* want to include those in the tests to be run. You can do so as follows by modifying the code under `if __name__ == "__main__":` to look like this:

```
if __name__ == "__main__":
    unittest_suite = unittest.TestLoader().loadTestsFromName("test_Biospam")
    doctest_suite = doctest.DocTestSuite(Biospam)
    suite = unittest.TestSuite((unittest_suite, doctest_suite))
    runner = unittest.TextTestRunner(sys.stdout, verbosity=2)
    runner.run(suite)
```

This is only relevant if you want to run the docstring tests when you execute `python test_Biospam.py` if it has some complex run-time dependency checking.

In general instead include the docstring tests by adding them to the `run_tests.py` as explained below.

23.3 Writing doctests

Python modules, classes and functions support built in documentation using docstrings. The doctest framework (included with Python) allows the developer to embed working examples in the docstrings, and have these examples automatically tested.

Currently only part of Biopython includes doctests. The `run_tests.py` script takes care of running the doctests. For this purpose, at the top of the `run_tests.py` script is a manually compiled list of modules to skip, important where optional external dependencies which may not be installed (e.g. the Reportlab and NumPy libraries). So, if you've added some doctests to the docstrings in a Biopython module, in order to have them excluded in the Biopython test suite, you must update `run_tests.py` to include your module. Currently, the relevant part of `run_tests.py` looks as follows:

```
# Following modules have historic failures. If you fix one of these
# please remove here!
EXCLUDE_DOCTEST_MODULES = [
    "Bio.PDB",
    "Bio.PDB.AbstractPropertyMap",
```

```
    "Bio.Phylo.Applications._Fasttree",
    "Bio.Phylo._io",
    "Bio.Phylo.TreeConstruction",
    "Bio.Phylo._utils",
]

# Exclude modules with online activity
# They are not excluded by default, use --offline to exclude them
ONLINE_DOCTEST_MODULES = ["Bio.Entrez", "Bio.ExPASy", "Bio.TogoWS"]

# Silently ignore any doctests for modules requiring numpy!
if numpy is None:
    EXCLUDE_DOCTEST_MODULES.extend(
        [
            "Bio.Affy.CelFile",
            "Bio.Cluster",
            # ...
        ]
    )
```

Note that we regard doctests primarily as documentation, so you should stick to typical usage. Generally complicated examples dealing with error conditions and the like would be best left to a dedicated unit test.

Note that if you want to write doctests involving file parsing, defining the file location complicates matters. Ideally use relative paths assuming the code will be run from the `Tests` directory, see the `Bio.SeqIO` doctests for an example of this.

To run the docstring tests only, use

```
$ python run_tests.py doctest
```

Note that the doctest system is fragile and care is needed to ensure your output will match on all the different versions of Python that Biopython supports (e.g. differences in floating point numbers).

23.4 Writing doctests in the Tutorial

This Tutorial you are reading has a lot of code snippets, which are often formatted like a doctest. We have our own system in file `test_Tutorial.py` to allow tagging code snippets in the Tutorial source to be run as Python doctests. This works by adding special `%doctest` comment lines before each Python block, e.g.

```
%doctest
\begin{minted}{pycon}
>>> from Bio.Seq import Seq
>>> s = Seq("ACGT")
>>> len(s)
4
\end{minted}
```

Often code examples are not self-contained, but continue from the previous Python block. Here we use the magic comment `%cont-doctest` as shown here:

```
%cont-doctest
\begin{minted}{pycon}
```

```
>>> s == "ACGT"
True
\end{minted}
```

The special `%doctest` comment line can take a working directory (relative to the `Doc/` folder) to use if you have any example data files, e.g. `%doctest examples` will use the `Doc/examples` folder, while `%doctest ../Tests/GenBank` will use the `Tests/GenBank` folder.

After the directory argument, you can specify any Python dependencies which must be present in order to run the test by adding `lib:XXX` to indicate `import XXX` must work, e.g. `%doctest examples lib:numpy`

You can run the Tutorial doctests via:

```
$ python test_Tutorial.py
```

or:

```
$ python run_tests.py test_Tutorial.py
```

Chapter 24

Where to go from here – contributing to Biopython

24.1 Bug Reports + Feature Requests

Getting feedback on the Biopython modules is very important to us. Open-source projects like this benefit greatly from feedback, bug-reports (and patches!) from a wide variety of contributors.

The main forums for discussing feature requests and potential bugs are the Biopython mailing list and issues or pull requests on GitHub.

Additionally, if you think you've found a new bug, you can submit it to our issue tracker at `https://github.com/biopython/biopython/issues` (this replaced the older Open Bioinformatics Foundation hosted RedMine tracker). This way, it won't get buried in anyone's Inbox and forgotten about.

24.2 Mailing lists and helping newcomers

We encourage all our uses to sign up to the main Biopython mailing list. Once you've got the hang of an area of Biopython, we'd encourage you to help answer questions from beginners. After all, you were a beginner once.

24.3 Contributing Documentation

We're happy to take feedback or contributions - either via a bug-report or on the Mailing List. While reading this tutorial, perhaps you noticed some topics you were interested in which were missing, or not clearly explained. There is also Biopython's built in documentation (the docstrings, these are also online), where again, you may be able to help fill in any blanks.

24.4 Contributing cookbook examples

As explained in Chapter 22, Biopython now has a wiki collection of user contributed "cookbook" examples, `http://biopython.org/wiki/Category:Cookbook` – maybe you can add to this?

24.5 Maintaining a distribution for a platform

We currently provide source code archives (suitable for any OS, if you have the right build tools installed), and pre-compiled wheels via `https://github.com/biopython/biopython-wheels` to cover the major operating

systems.

Most major Linux distributions have volunteers who take these source code releases, and compile them into packages for Linux users to easily install (taking care of dependencies etc). This is really great and we are of course very grateful. If you would like to contribute to this work, please find out more about how your Linux distribution handles this. There is a similar process for conda packages via https://github.com/conda-forge/biopython-feedstock thanks to the conda-forge team.

Below are some tips for certain platforms to maybe get people started with helping out:

Windows – You must first make sure you have a C compiler on your Windows computer, and that you can compile and install things (this is the hard bit - see the Biopython installation instructions for info on how to do this).

RPMs – RPMs are pretty popular package systems on some Linux platforms. There is lots of documentation on RPMs available at http://www.rpm.org to help you get started with them. To create an RPM for your platform is really easy. You just need to be able to build the package from source (having a C compiler that works is thus essential) – see the Biopython installation instructions for more info on this.

To make the RPM, you just need to do:

```
$ python setup.py bdist_rpm
```

This will create an RPM for your specific platform and a source RPM in the directory `dist`. This RPM should be good and ready to go, so this is all you need to do! Nice and easy.

Macintosh – Since Apple moved to Mac OS X, things have become much easier on the Mac. We generally treat it as just another Unix variant, and installing Biopython from source is just as easy as on Linux. The easiest way to get all the GCC compilers etc installed is to install Apple's X-Code. We might be able to provide click and run installers for Mac OS X, but to date there hasn't been any demand.

Once you've got a package, please test it on your system to make sure it installs everything in a good way and seems to work properly. Once you feel good about it, make a pull request on GitHub and write to our Biopython mailing list. You've done it. Thanks!

24.6 Contributing Unit Tests

Even if you don't have any new functionality to add to Biopython, but you want to write some code, please consider extending our unit test coverage. We've devoted all of Chapter 23 to this topic.

24.7 Contributing Code

There are no barriers to joining Biopython code development other than an interest in creating biology-related code in Python. The best place to express an interest is on the Biopython mailing lists – just let us know you are interested in coding and what kind of stuff you want to work on. Normally, we try to have some discussion on modules before coding them, since that helps generate good ideas – then just feel free to jump right in and start coding!

The main Biopython release tries to be fairly uniform and interworkable, to make it easier for users. You can read about some of (fairly informal) coding style guidelines we try to use in Biopython in the contributing documentation at http://biopython.org/wiki/Contributing. We also try to add code to the distribution along with tests (see Chapter 23 for more info on the regression testing framework) and documentation, so that everything can stay as workable and well documented as possible (including docstrings). This is, of course, the most ideal situation, under many situations you'll be able to find other people on the list who

will be willing to help add documentation or more tests for your code once you make it available. So, to end this paragraph like the last, feel free to start working!

Please note that to make a code contribution you must have the legal right to contribute it and license it under the Biopython license. If you wrote it all yourself, and it is not based on any other code, this shouldn't be a problem. However, there are issues if you want to contribute a derivative work - for example something based on GPL or LPGL licensed code would not be compatible with our license. If you have any queries on this, please discuss the issue on the mailing list or GitHub.

Another point of concern for any additions to Biopython regards any build time or run time dependencies. Generally speaking, writing code to interact with a standalone tool (like BLAST, EMBOSS or ClustalW) doesn't present a big problem. However, any dependency on another library - even a Python library (especially one needed in order to compile and install Biopython like NumPy) would need further discussion.

Additionally, if you have code that you don't think fits in the distribution, but that you want to make available, we maintain Script Central (http://biopython.org/wiki/Scriptcentral) which has pointers to freely available code in Python for bioinformatics.

Hopefully this documentation has got you excited enough about Biopython to try it out (and most importantly, contribute!). Thanks for reading all the way through!

Chapter 25

Appendix: Useful stuff about Python

If you haven't spent a lot of time programming in Python, many questions and problems that come up in using Biopython are often related to Python itself. This section tries to present some ideas and code that come up often (at least for us!) while using the Biopython libraries. If you have any suggestions for useful pointers that could go here, please contribute!

25.1 What the heck is a handle?

Handles are mentioned quite frequently throughout this documentation, and are also fairly confusing (at least to me!). Basically, you can think of a handle as being a "wrapper" around text information.

Handles provide (at least) two benefits over plain text information:

1. They provide a standard way to deal with information stored in different ways. The text information can be in a file, or in a string stored in memory, or the output from a command line program, or at some remote website, but the handle provides a common way of dealing with information in all of these formats.

2. They allow text information to be read incrementally, instead of all at once. This is really important when you are dealing with huge text files which would use up all of your memory if you had to load them all.

Handles can deal with text information that is being read (e. g. reading from a file) or written (e. g. writing information to a file). In the case of a "read" handle, commonly used functions are `read()`, which reads the entire text information from the handle, and `readline()`, which reads information one line at a time. For "write" handles, the function `write()` is regularly used.

The most common usage for handles is reading information from a file, which is done using the built-in Python function `open`. Here, we handle to the file `m_cold.fasta` which you can download here (or find included in the Biopython source code as `Doc/examples/m_cold.fasta`).

```
>>> handle = open("m_cold.fasta", "r")
>>> handle.readline()
">gi|8332116|gb|BE037100.1|BE037100 MP14H09 MP Mesembryanthemum ...\n"
```

Handles are regularly used in Biopython for passing information to parsers. For example, since Biopython 1.54 the main functions in `Bio.SeqIO` and `Bio.AlignIO` have allowed you to use a filename instead of a handle:

```
from Bio import SeqIO
```

```
for record in SeqIO.parse("m_cold.fasta", "fasta"):
    print(record.id, len(record))
```

On older versions of Biopython you had to use a handle, e.g.

```
from Bio import SeqIO

handle = open("m_cold.fasta", "r")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()
```

This pattern is still useful - for example suppose you have a gzip compressed FASTA file you want to parse:

```
import gzip
from Bio import SeqIO

handle = gzip.open("m_cold.fasta.gz", "rt")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record))
handle.close()
```

With our parsers for plain text files, it is essential to use gzip in text mode (the default is binary mode). See Section 5.2 for more examples like this, including reading bzip2 compressed files.

25.1.1 Creating a handle from a string

One useful thing is to be able to turn information contained in a string into a handle. The following example shows how to do this using `StringIO` from the Python standard library:

```
>>> my_info = "A string\n with multiple lines."
>>> print(my_info)
A string
 with multiple lines.
>>> from io import StringIO
>>> my_info_handle = StringIO(my_info)
>>> first_line = my_info_handle.readline()
>>> print(first_line)
A string
<BLANKLINE>
>>> second_line = my_info_handle.readline()
>>> print(second_line)
 with multiple lines.
```

Bibliography

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, David J. Lipman: "Basic Local Alignment Search Tool". *Journal of Molecular Biology* **215** (3): 403–410 (1990). https://doi.org/10.1016/S0022-2836%2805%2980360-2.

[2] Timothy L. Bailey and Charles Elkan: "Fitting a mixture model by expectation maximization to discover motifs in biopolymers", *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology* 28–36. AAAI Press, Menlo Park, California (1994).

[3] Douglas R. Cavener: "Comparison of the consensus sequence flanking translational start sites in Drosophila and vertebrates." *Nucleic Acids Research* **15** (4): 1353–1361 (1987). https://doi.org/10.1093/nar/15.4.1353

[4] Brad Chapman and Jeff Chang: "Biopython: Python tools for computational biology". *ACM SIGBIO Newsletter* **20** (2): 15–19 (August 2000).

[5] Peter J. A. Cock, Tiago Antao, Jeffrey T. Chang, Brad A. Chapman, Cymon J. Cox, Andrew Dalke, Iddo Friedberg, Thomas Hamelryck, Frank Kauff, Bartek Wilczynski, Michiel J. L. de Hoon: "Biopython: freely available Python tools for computational molecular biology and bioinformatics". *Bioinformatics* **25** (11), 1422–1423 (2009).

[6] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, Peter M. Rice: "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants". *Nucleic Acids Research* **38** (6): 1767–1771 (2010). https://doi.org/10.1093/nar/gkp1137 https://doi.org/10.1093/bioinformatics/btp163

[7] Athel Cornish-Bowden: "Nomenclature for incompletely specified bases in nucleic acid sequences: Recommendations 1984." *Nucleic Acids Research* **13** (9): 3021–3030 (1985). https://doi.org/10.1093/nar/13.9.3021

[8] Aaron E. Darling, Bob Mau, Frederick R. Blattner, Nicole T. Perna: "Mauve: Multiple alignment of conserved genomic sequence with rearrangements." *Genome Research* **14** (7): 1394–1403 (2004). https://doi.org/10.1101/gr.2289704

[9] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt: "A Model of Evolutionary Change in Proteins." *Atlas of Protein Sequence and Structure*, Volume 5, Supplement 3, 1978: 345–352. The National Biomedical Research Foundation, 1979.

[10] Michiel J. L. de Hoon, Seiya Imoto, John Nolan, Satoru Miyano: "Open source clustering software". *Bioinformatics* **20** (9): 1453–1454 (2004). https://doi.org/10.1093/bioinformatics/bth078

[11] Richard Durbin, Sean R. Eddy, Anders Krogh, Graeme Mitchison: "Biological sequence analysis: Probabilistic models of proteins and nucleic acids". Cambridge University Press, Cambridge, UK (1998).

[12] Michiel B. Eisen, Paul T. Spellman, Patrick O. Brown, David Botstein: "Cluster analysis and display of genome-wide expression patterns". *Proceedings of the National Academy of Sciences USA* **95** (25): 14863–14868 (1998). https://doi.org/10.1073/pnas.96.19.10943-c

[13] Nick Goldman and Ziheng Yang: "A codon-based model of nucleotide substitution for protein-coding DNA sequences." *Molecular Biology and Evolution* **11** (5) 725–736 (1994). https://doi.org/10.1093/oxfordjournals.molbev.a040153.

[14] Gene H. Golub, Christian Reinsch: "Singular value decomposition and least squares solutions". In *Handbook for Automatic Computation*, **2**, (Linear Algebra) (J. H. Wilkinson and C. Reinsch, eds), 134–151. New York: Springer-Verlag (1971).

[15] Gene H. Golub, Charles F. Van Loan: *Matrix computations*, 2nd edition (1989).

[16] Thomas Hamelryck and Bernard Manderick: "PDB parser and structure class implemented in Python". *Bioinformatics* **19** (17): 2308–2310 (2003) https://doi.org/10.1093/bioinformatics/btg299.

[17] Thomas Hamelryck: "Efficient identification of side-chain patterns using a multidimensional index tree". *Proteins* **51** (1): 96–108 (2003). https://doi.org/10.1002/prot.10338

[18] Thomas Hamelryck: "An amino acid has two sides; A new 2D measure provides a different view of solvent exposure". *Proteins* **59** (1): 29–48 (2005). https://doi.org/10.1002/prot.20379.

[19] Steven Henikoff, Jorja G. Henikoff: "Amino acid substitution matrices from protein blocks." *Proceedings of the National Academy of Sciences USA* **89** (2): 10915–10919 (1992). https://doi.org/10.1073/pnas.89.22.10915.

[20] Yukako Hihara, Ayako Kamei, Minoru Kanehisa, Aaron Kaplan and Masahiko Ikeuchi: "DNA microarray analysis of cyanobacterial gene expression during acclimation to high light". *Plant Cell* **13** (4): 793–806 (2001). https://doi.org/10.1105/tpc.13.4.793.

[21] Richard Hughey, Anders Krogh: "Hidden Markov models for sequence analysis: extension and analysis of the basic method". *Computer Applications in the Biosciences: CABIOS* **12** (2): 95–107 (1996). https://doi.org/10.1093/bioinformatics/12.2.95

[22] Florian Jupe, Leighton Pritchard, Graham J. Etherington, Katrin MacKenzie, Peter JA Cock, Frank Wright, Sanjeev Kumar Sharma1, Dan Bolser, Glenn J Bryan, Jonathan DG Jones, Ingo Hein: "Identification and localisation of the NB-LRR gene family within the potato genome". *BMC Genomics* **13**: 75 (2012). https://doi.org/10.1186/1471-2164-13-75

[23] Voratas Kachitvichyanukul, Bruce W. Schmeiser: Binomial Random Variate Generation. *Communications of the ACM* **31** (2): 216–222 (1988). https://doi.org/10.1145/42372.42381

[24] W. James Kent: "BLAT — The BLAST-Like Alignment Tool". *Genome Research* **12**: 656–664 (2002). https://doi.org/10.1101/gr.229202

[25] Teuvo Kohonen: "Self-organizing maps", 2nd Edition. Berlin; New York: Springer-Verlag (1997).

[26] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjölander, David Haussler: "Hidden Markov Models in computational biology: Applications to protein modeling." *Journal of Molecular Biology* **235** (5): 1501–1531 (1994). https://doi.org/10.1006/jmbi.1994.1104

[27] Pierre L'Ecuyer: "Efficient and Portable Combined Random Number Generators." *Communications of the ACM* **31** (6): 742–749,774 (1988). https://doi.org/10.1145/62959.62969

[28] Wen-Hsiung Li, Chung-I Wu, Chi-Cheng Luo: "A new method for estimating synonymous and nonsynonymous rates of nucleotide substitution considering the relative likelihood of nucleotide and codon changes." *Molecular Biology and Evolution* **2** (2): 150–174 (1985). https://doi.org/10.1093/oxfordjournals.molbev.a040343

[29] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin: "The Sequence Alignment/Map format and SAMtools." *Bioinformatics* **25** (16): 2078–2079 (2009). https://doi.org/10.1093/bioinformatics/btp352

[30] David R. Maddison, David L. Swofford, Wayne P. Maddison: "Nexus: An Extensible File Format for Systematic Information." *Systematic Biology* **46** (4): 590–621 (1997). https://doi.org/10.1093/sysbio/46.4.590.

[31] Indraneel Majumdar, S. Sri Krishna, Nick V. Grishin: "PALSSE: A program to delineate linear secondary structural elements from protein structures." *BMC Bioinformatics* **6**: 202 (2005). https://doi.org/10.1186/1471-2105-6-202.

[32] V. Matys, E. Fricke, R. Geffers, E. Gössling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A.E. Kel, O.V. Kel-Margoulis, D.U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Münch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, E. Wingender E: "TRANSFAC: transcriptional regulation, from patterns to profiles." Nucleic Acids Research **31** (1): 374–378 (2003). https://doi.org/10.1093/nar/gkg108

[33] Masatoshi Nei and Takashi Gojobori: "Simple methods for estimating the numbers of synonymous and nonsynonymous nucleotide substitutions." *Molecular Biology and Evolution* **3** (5): 418–426 (1986). https://doi.org/10.1093/oxfordjournals.molbev.a040410

[34] William R. Pearson, David J. Lipman: "Improved tools for biological sequence comparison." *Proceedings of the National Academy of Sciences USA* **85** (8): 2444–2448 (1988). https://doi.org/10.1073/pnas.85.8.2444

[35] Leighton Pritchard, Jennifer A. White, Paul R.J. Birch, Ian K. Toth: "GenomeDiagram: a python package for the visualization of large-scale genomic data". *Bioinformatics* **22** (5): 616–617 (2006). https://doi.org/10.1093/bioinformatics/btk021

[36] Caroline Proux, Douwe van Sinderen, Juan Suarez, Pilar Garcia, Victor Ladero, Gerald F. Fitzgerald, Frank Desiere, Harald Brüssow: "The dilemma of phage taxonomy illustrated by comparative genomics of Sfi21-Like Siphoviridae in lactic acid bacteria". *Journal of Bacteriology* **184** (21): 6026–6036 (2002). https://doi.org/10.1128/JB.184.21.6026-6036.2002

[37] Peter Rice, Ian Longden, Alan Bleasby: "EMBOSS: The European Molecular Biology Open Software Suite." *Trends in Genetics* **16** (6): 276–277 (2000). https://doi.org/10.1016/S0168-9525(00)02024-2

[38] Alok Saldanha: "Java Treeview—extensible visualization of microarray data". *Bioinformatics* **20** (17): 3246–3248 (2004). https://doi.org/10.1093/bioinformatics/bth349

[39] Thomas D. Schneider, Gary D. Stormo, Larry Gold: "Information content of binding sites on nucleotide sequences". *Journal of Molecular Biology* **188** (3): 415–431 (1986). https://doi.org/10.1016/0022-2836(86)90165-8

[40] Adrian Schneider, Gina M. Cannarozzi, and Gaston H. Gonnet: "Empirical codon substitution matrix". *BMC Bioinformatics* **6**: 134 (2005). https://doi.org/10.1186/1471-2105-6-134

[41] Robin Sibson: "SLINK: An optimally efficient algorithm for the single-link cluster method". *The Computer Journal* **16** (1): 30–34 (1973). https://doi.org/10.1093/comjnl/16.1.30

[42] Guy St C. Slater, Ewan Birney: "Automated generation of heuristics for biological sequence comparison." *BMC Bioinformatics* **6**: 31 (2005). https://doi.org/10.1186/1471-2105-6-31

[43] George W. Snedecor, William G. Cochran: *Statistical methods*. Ames, Iowa: Iowa State University Press (1989).

[44] Martin Steinegger, Markus Meier, Milot Mirdita, Harald Vöhringer, Stephan J. Haunsberger, Johannes Söding: "HH-suite3 for fast remote homology detection and deep protein annotation." *BMC Bioinformatics* **20**: 473 (2019). https://doi.org/10.1186/s12859-019-3019-7

[45] Eric Talevich, Brandon M. Invergo, Peter J.A. Cock, Brad A. Chapman: "Bio.Phylo: A unified toolkit for processing, analyzing and visualizing phylogenetic trees in Biopython". *BMC Bioinformatics* **13**: 209 (2012). https://doi.org/10.1186/1471-2105-13-209

[46] Pablo Tamayo, Donna Slonim, Jill Mesirov, Qing Zhu, Sutisak Kitareewan, Ethan Dmitrovsky, Eric S. Lander, Todd R. Golub: "Interpreting patterns of gene expression with self-organizing maps: Methods and application to hematopoietic differentiation". *Proceedings of the National Academy of Sciences USA* **96** (6): 2907–2912 (1999). https://doi.org/10.1073/pnas.96.6.2907

[47] Ian K. Toth, Leighton Pritchard, Paul R. J. Birch: "Comparative genomics reveals what makes an enterobacterial plant pathogen". *Annual Review of Phytopathology* **44**: 305–336 (2006). https://doi.org/10.1146/annurev.phyto.44.070505.143444

[48] John W. Tukey: "Exploratory data analysis". Reading, Mass.: Addison-Wesley Pub. Co. (1977).

[49] Géraldine A. van der Auwera, Jaroslaw E. Król, Haruo Suzuki, Brian Foster, Rob van Houdt, Celeste J. Brown, Max Mergeay, Eva M. Top: "Plasmids captured in C. metallidurans CH34: defining the PromA family of broad-host-range plasmids". *Antonie van Leeuwenhoek* **96** (2): 193–204 (2009). https://doi.org/10.1007/s10482-009-9316-9

[50] Michael S. Waterman, Mark Eggert: "A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons". *Journal of Molecular Biology* **197** (4): 723–728 (1987). https://doi.org/10.1016/0022-2836(87)90478-5

[51] Ziheng Yang and Rasmus Nielsen: "Estimating synonymous and nonsynonymous substitution rates under realistic evolutionary models". *Molecular Biology and Evolution* **17** (1): 32–43 (2000). https://doi.org/10.1093/oxfordjournals.molbev.a026236

[52] Ka Yee Yeung, Walter L. Ruzzo: "Principal Component Analysis for clustering gene expression data". *Bioinformatics* **17** (9): 763–774 (2001). https://doi.org/10.1093/bioinformatics/17.9.763