
biopython_cn Documentation

Release 0.1

biopythoners

January 23, 2014

1	第 1 章介绍	3
1.1	什么是 Biopython?	3
1.2	在 Biopython 包中我能发现什么?	3
1.3	安装 Biopython	4
1.4	常见问答 (FAQ)	4
2	第 2 章快速开始——你能用 Biopython 做什么?	9
2.1	Biopython 功能概览	9
2.2	处理序列	9
2.3	用法示例	10
2.4	解析序列文件格式	10
2.5	连接生物学数据库	12
2.6	下一步做什么	12
3	第 3 章生物序列对象	13
3.1	序列和字母表	13
3.2	序列表现的像字符串一样	14
3.3	切取序列	15
3.4	将序列对象转换成字符串	16
3.5	连接或添加序列	16
3.6	改变大小写	17
3.7	核苷酸序列和 (反向) 互补序列	17
3.8	转录	18
3.9	翻译	20
3.10	翻译表	22
3.11	比较 Seq 对象	23
3.12	MutableSeq 对象	24
3.13	UnknownSeq 对象	25
3.14	直接使用字符串	26
4	第 4 章序列注释对象	27
4.1	SeqRecord 对象	27
4.2	创建 SeqRecord	28
4.3	Feature, location 和 position 对象	31
4.4	References	34
4.5	格式化方法	34
4.6	SeqRecord 切片	35
4.7	SeqRecord 对象相加	37
4.8	反向互补 SeqRecord 对象	39
5	第 5 章序列输入和输出	41

5.1 解析/读取序列	41
5.2 从压缩文档读取解析序列信息	45
5.3 解析来自网络的序列	46
5.4 序列文件作为字典	48
5.5 写入序列文件	54
6 第 6 章多序列比对	59
6.1 读取多序列比对数据	59
6.2 序列比对的写出	65
6.3 序列比对的操纵	69
6.4 构建序列比对的工具	71
7 第 7 章 BLAST	79
7.1 通过 Internet 运行 BLAST	79
7.2 本地运行 BLAST	81
7.3 解析 BLAST 输出	82
7.4 BLAST 记录类	84
7.5 废弃的 BLAST 解析器	87
7.6 处理 PSI-BLAST	90
7.7 处理 RPS-BLAST	90
8 第 8 章 BLAST 和其他序列搜索工具 (实验性质的代码)	91
8.1 SearchIO 对象模型	91
8.2 一个关于标准和惯例的注意事项	102
8.3 读取搜索输出文件	103
8.4 用索引处理含有大量搜索输出的文件	104
8.5 写入和转换搜索输出文件	104
9 第 9 章访问 NCBI Entrez 数据库	107
9.1 Entrez 简介	108
9.2 EInfo: 获取 Entrez 数据库的信息	108
9.3 ESearch: 搜索 Entrez 数据库	110
9.4 EPost: 上传 identifiers 的列表	111
9.5 ESummary: 通过主要的 IDs 来获取摘要	112
9.6 EFetch: 从 Entrez 下载更多的记录	112
9.7 ELink: 在 NCBI Entrez 中搜索相关的条目	115
9.8 EGQuery: 全局搜索 - 统计搜索的条目	116
9.9 ESpell: 获得拼写建议	116
9.10 解析大的 Entrez XML 文件	117
9.11 错误处理	117
9.12 专用的解析器	119
9.13 使用代理	123
9.14 实例	124
9.15 使用历史记录和 WebEnv	129
10 第 10 章 Swiss-Prot 和 ExPASy	133
10.1 解析 Swiss-Prot 文件	133
10.2 解析 Prosite 记录	136
10.3 解析 Prosite 文件记录	137
10.4 解析酶记录	137
10.5 Accessing the ExPASy server	138
10.6 浏览 Prosite 数据库	140
11 第 11 章走向 3D : PDB 模块	143
11.1 晶体结构文件的读与写	143

11.2	结构的表示	145
11.3	紊乱	150
11.4	异质残基	151
11.5	浏览 Structure 对象	152
11.6	分析结构	155
11.7	PDB 文件中的常见问题	157
11.8	访问 Protein Data Bank	159
11.9	常见问题	160
12	第 12 章 Bio.PopGen : 群体遗传学	163
12.1	GenePop	163
12.2	溯祖模拟 (Coalescent simulation)	164
12.3	其它应用程序	168
12.4	未来发展	170
13	第 13 章 Bio.Phylo 系统发育分析	171
13.1	示例: 树中有什么?	171
13.2	I/O 函数	176
13.3	查看和导出树	176
13.4	使用 Tree 和 Clade 对象	182
13.5	运行外部程序	185
13.6	PAML 整合	185
13.7	未来计划	186
14	第 14 章使用 Bio.motifs 进行模体序列分析	187
14.1	模体对象	187
14.2	位置权重矩阵	197
14.3	位置特异性得分矩阵	198
14.4	搜索实例	199
14.5	模体对象自身相关的位置特异性得分矩阵	201
14.6	模体比较	203
14.7	查找 <i>De novo</i> 模体	204
14.8	相关链接	205
14.9	旧版 Bio.Motif 模块	206
15	第 15 章聚类分析	213
	数据表示法	213
	缺失值	213
	随机数生成器	213
15.1	距离函数	214
15.2	计算类的相关性质	218
15.3	划分算法	219
15.4	系统聚类	221
15.5	Self-Organizing Maps	224
15.6	主成分分析	226
15.7	处理 Cluster/TreeView-type 文件	226
15.8	示例	230
15.9	附加函数	231
16	第 16 章监督学习方法	233
16.1	Logistic 回归模型	233
16.2	k -最近邻居法 (KNN)	238
16.3	Naive 贝叶斯	241
16.4	最大熵	241
16.5	马尔科夫模型	241

17 第 17 章 Graphics 模块中的基因组可视化包—GenomeDiagram	243
17.1 基因组可视化包—GenomeDiagram	243
17.2 染色体	262
18 第 18 章 Cookbook —用它做一些很酷的事情	267
18.1 操作序列文件	267
18.2 序列解析与简单作图	279
18.3 处理序列比对	285
18.4 替换矩阵	289
18.5 BioSQL —存储序列到关系数据库中	290
19 第 19 章 Biopython 测试框架	291
19.1 运行测试	291
19.2 编写测试	292
19.3 编写 doctests	296
20 第 20 章高级	297
20.1 解析器的设计	297
20.2 替换矩阵	297
21 第 21 章为 Biopython 做贡献	301
21.1 错误报告与功能需求	301
21.2 邮件列表与帮助新手	301
21.3 贡献文档	301
21.4 贡献学习手册示例	301
21.5 维护跨平台发行版本	302
21.6 贡献单元测试 (Unit Tests)	302
21.7 贡献源码	302
22 第 22 章附录 : Python 相关知识	305
22.1 到底什么是句柄 (handle) ?	305
23 References	307
24 Indices and tables	311

Contents:

第 1 章介绍

1.1 什么是 Biopython ?

Biopython 工程是一个使用 Python 来开发计算分子生物学工具的国际团体。(<http://www.python.org>) Python 是一种面向对象的、解释型的、灵活的语言，在计算机科学中日益流行。Python 易学，语法明晰，并且能很容易的使用以 C，C++ 或者 FORTRAN 编写的模块实现扩展。

Biopython 官网 (<http://www.biopython.org>) 为使用和研究生物信息学的开发者提供了一个在线的资源库，包括模块、脚本以及一些基于 Python 的软件的网站链接。一般来讲，Biopython 致力于通过创造高质量的和可重复利用的模块及类，从而使得 Python 在生物信息学中的应用变得更加容易。Biopython 的特点包括解析各种生物信息学格式的文件 (BLAST, Clustalw, FASTA, Genbank...), 访问在线的服务器 (NCBI, Expasy...), 常见和不那么常见程序的接口 (Clustalw, DSSP, MSMS...), 标准的序列类，各种收集的模块，KD 树数据结构等等，还有一些文档。

基本来说，我们喜欢使用 Python 来编程，并且希望通过创建高质量、可复用的模块和脚本来使得 Python 在生物信息学中的应用变得容易。

1.2 在 Biopython 包中我能发现什么？

主要的 Biopython 发行版本有很多种功能，包括：

- 将生物信息学文件解析为 Python 可用的数据结构，包含以下支持的格式：
 - Blast 输出结果— standalone 和在线 Blast
 - Clustalw
 - FASTA
 - GenBank
 - PubMed 和 Medline
 - ExPASy 文件, 如 Enzyme 和 Prosite
 - SCOP, 包括'dom'和'lin'文件
 - UniGene
 - SwissProt
- 被支持格式的文件可以通过记录来重复或者通过字典界面来索引。
- 处理常见的生物信息学在线数据库的代码：
 - NCBI – Blast, Entrez 和 PubMed 服务

- ExPASy – Swiss-Prot 和 Prosite 条目, 包括 Prosite 搜索
- 常见生物信息学程序的接口, 例如:
 - NCBI 的 Standalone Blast
 - Clustalw 比对程序
 - EMBOSS 命令行工具
- 一个能处理序列、ID 和序列特征的标准序列类。
- 对序列实现常规操作的工具, 如翻译, 转录和权重计算。
- 利用 k 最近邻接、Bayes 或 SVM 对数据进行分类的代码。
- 处理比对的代码, 包括创建和处理替换矩阵的标准方法。
- 分发并行任务到不同进程的代码。
- 实现序列的基本操作, 翻译以及 BLAST 等功能的 GUI 程序。
- 使用这些模块的详细文档和帮助, 包括此文件, 在线的 wiki 文档, 网站和邮件列表。
- 整合 BioSQL, 一个也被 BioPerl 和 BioJava 支持的数据库架构。

我们希望这些能给你足够的理由去下载并开始使用 Biopython !

1.3 安装 Biopython

Biopython 的所有安装信息在此文档中分开, 以便于更容易保持更新。

简短的版本去我们的下载页面 (<http://biopython.org/wiki/Download>), 下载并安装所列举的 dependencies, 然后下载并安装 Biopython。Biopython 能在多种平台上运行 (Windows, Mac, 各种版本的 Linux 和 Unix)。对于 Windows 我们提供预编译的一键式安装程序, 而对 Unix 和其他操作系统, 你必须按照附带的 README 文件从源开始安装。这通常是很简单的, 只要标准命令:

```
python setup.py build
python setup.py test
sudo python setup.py install
```

(事实上你可以跳过 build 和 test, 直接 install。但最好是确保所有的东西看起来都没问题。)

我们的安装说明的详细版本包括了 Python 的安装, Biopython dependencies 的安装以及 Biopython 本身的安装。可从 PDF (<http://biopython.org/DIST/docs/install/Installation.pdf>) 和 HTML 格式获得。(<http://biopython.org/DIST/docs/install/Installation.html>)。

1.4 常见问答 (FAQ)

1. 在科学出版中我怎样引用 *Biopython* ?

请引用我们的应用笔记 [1, Cock *et al.*, 2009] 作为主要的 Biopython 参考文献。另外, 如果可以, 请引用以下任意出版物, 特别是作为 Biopython 特定模块的参考文献的话。(更多信息可在我们网站上获得):

- 对于官方项目声明: [13, Chapman and Chang, 2000];
- 对于 Bio.PDB: [18, Hamelryck and Manderick, 2003];
- 对于 Bio.Cluster: [14, De Hoon *et al.*, 2004];

- 对于 `Bio.Graphics.GenomeDiagram`: [2, Pritchard *et al.*, 2006];
- 对于 `Bio.Phylo` 和 `Bio.Phylo.PAML`: [9, Talevich *et al.*, 2012];
- 对于在 Biopython, BioPerl, BioRuby, BioJava 和 EMBOSS 支持的 FASTQ 格式文件: [7, Cock *et al.*, 2010].

2. 我该怎样以大写字母写“*Biopython*”? 写成“*BioPython*”可以吗?

正确的大写是“Biopython”而不是“BioPython”(虽然对于 BioPerl, BioRuby 和 BioJava 是这样)。

3. 我怎样查看自己安装的 *Biopython* 的版本?

使用以下代码:

```
>>> import Bio
>>> print Bio.__version__
...
```

如果“import Bio”这行报错, 说明 Biopython 未被安装。如果第二行报错, 你的版本已经很过时了。如果版本号以“+”号结束, 说明你用的并不是官方版本, 而是开发代码的快照。

4. 此文档的最新版本在哪里?

如果你下载的是一个 Biopython 源代码包, 那么它将包含此文档 HTML 和 PDF 两种格式的相应版本。此文档最新出版的版本可通过在线获得(每个版本的更新):

- <http://biopython.org/DIST/docs/tutorial/Tutorial.html>
- <http://biopython.org/DIST/docs/tutorial/Tutorial.pdf>

如果你使用的是从我们库中获得的尚未发布的最新代码, 你可以在这里找到还在开发中的教程的拷贝:

- <http://biopython.org/DIST/docs/tutorial/Tutorial-dev.html>
- <http://biopython.org/DIST/docs/tutorial/Tutorial-dev.pdf>

5. 我需要哪一个“*Numerical Python*”?

对于 Biopython 1.48 或更早的版本, 你需要老的 Numeric 模块。对于 Biopython 1.49 及更高的版本, 你需要更新的 NumPy 来代替。Numeric 和 NumPy 都可以在同一台机器上安装。也可以访问: <http://numpy.scipy.org/>

6. 为什么 Seq 对象缺少了这篇教程里的(反向) *transcription* 和 *translation* 方法?

你需要 Biopython 1.49 或更新的版本。或者, 使用以下 3.14 部分中的 `Bio.Seq` 模块功能。

7. 为什么 Seq 对象缺少了这篇教程中的 *upper* 和 *lower* 方法?

你需要 Biopython 1.53 或更新版本。或者, 使用 `str(my_seq).upper()` 来获得大写字符串。如果你需要一个 Seq 对象, 试试 `Seq(str(my_seq).upper())`, 但是要小心重用相同的字母。

8. 为什么 Seq 对象的 *translation* 方法不支持本教程中描述的 *cds* 选项?

你需要 Biopython 1.51 或更新版本。

9. 为什么 `Bio.SeqIO` 不能正常工作? 它导入正常但是没有解析函数等。

你需要 Biopython 1.43 或更新版本。较老的版本确实包含了一些相关的代码在 `Bio.SeqIO` 下面但是后来就被移除了——这就是为什么 `import` 是正常的。

10. 为什么 `Bio.SeqIO.read()` 不能正常工作? 该模块导入正常但是并没有 *read* 函数!

你需要 Biopython 1.45 或更新的版本。或者, 使用 `Bio.SeqIO.parse(...).next()` 来代替。

11. 为什么没有 `Bio.AlignIO` ? 模块导入失败!

你需要 Biopython 1.46 或更新的版本。

12. `Bio.SeqIO` 和 `Bio.AlignIO` 读写什么样的文件格式?

请检查内建文档 (`from Bio import SeqIO`, 然后 `help(SeqIO)`), 或见 wiki 上的最新条目: <http://biopython.org/wiki/SeqIO> 以及 <http://biopython.org/wiki/AlignIO>

13. 为什么 `Bio.SeqIO` 和 `Bio.AlignIO` 的 `input` 函数不让我提供一个序列字母?

你需要 Biopython 1.49 或更新版本。

14. 为什么 `Bio.SeqIO` 和 `Bio.AlignIO` 函数 `parse`, `read` 和 `write` 不能使用文件名? 它们坚持句柄!

你需要 Biopython 1.54 或更新的版本。或者明确使用句柄。(见 Section 22.1). 一定要记得当你写完数据后关闭输出句柄。

15. 为什么 `Bio.SeqIO.write()` 和 `Bio.AlignIO.write()` 函数不接受单个记录或比对? 它们坚持需要一个列表或迭代器!

你需要 Biopython 1.54 或更新版本, 或将该条目以 `[...]` 包起来形成一个单元素的列表。

16. 为什么 `str(...)` 不给我一个 `Seq` 对象的全序列?

你需要 Biopython 1.45 或更新的版本。或者, 与其使用 `str(my_seq)`, 不如试试 `my_seq.tostring()` 这也能在最近的 Biopython 版本上工作)。

17. 为什么 `Bio.Blast` 不能处理最新的 *NCBI blast* 输出文本文件结果?

NCBI 在不断的调整 BLAST 工具的纯文本输出, 导致我们的解析器需要不断更新。如果你没使用最新版本的 Biopython, 你可以试试升级。但是, 我们 (还有 NCBI) 推荐你使用 HTML 格式输出来代替, 因为 HTML 是设计给电脑程序读取的。

18. 为什么 `Bio.Entrez.read()` 不能正常工作? 模块导入正常但是没有 `read` 函数!

你需要 Biopython 1.46 或更新的版本。

19. 为什么 `Bio.Entrez.parse()` 不能正常工作? 模块导入正常但是没有 `parse` 函数!

你需要 Biopython 1.52 或更新的版本。

20. 为什么我的脚本使用了 `Bio.Entrez.efetch()` 便停止工作了?

这可能是由于 NCBI 在 2012 年 2 月引进 `EFetch 2.0` 后发生了改变。首先, 他们改变了默认的返回方式——你可能想添加 `retmode="text"` 到你的 call。其次, 他们对于怎么提供一个 ID 列表变得更加严格——Biopython 1.59 及之后版本或自动将一个列表转换成逗号分隔的字符串。

21. 为什么 `Bio.Blast.NCBIWWW.qblast()` 没有给出与 *NCBI BLAST* 网站上相同的结果?

你需要指定相同的选项——NCBI 经常调整网站上的默认设置, 并且他们不再匹配 QBLAST 的默认设置了。请检查 `gap` 罚分和期望值阈值。

22. 为什么 `Bio.Blast.NCBIXML.read()` 不正常工作? 模块导入了但是没有 `read` 函数!

你需要 Biopython 1.50 或更新的版本。或者, 使用 `Bio.Blast.NCBIXML.parse(...).next()` 代替。

23. 为什么我的 `SeqRecord` 对象没有一个 `letter_annotations` 的属性?

Per-letter-annotation 已经被加入到 Biopython 1.50 中。

24. 为什么我无法切片我的 `SeqRecord` 来获取一个子记录? 你需要 Biopython 1.50 或更新版本。

25. 为什么我无法一起添加 `SeqRecord` 对象?

你需要 Biopython 1.53 或更新版本。

26. 为什么 `Bio.SeqIO.convert()` 或 `Bio.AlignIO.convert()` 不能正常工作？模块导入正常但是没有 `convert` 函数！

你需要 Biopython 1.52 或更新版本。或者，按以下教程中描述的结合 `parse` 和 `write` 函数。（见 Sections 5.5.2 和 6.2.1）。

27. 为什么 `Bio.SeqIO.index()` 不能正常工作？模块导入正常但是没有 `index` 函数！

你需要 Biopython 1.52 或更新版本。

28. 为什么 `Bio.SeqIO.index.db()` 不能正常工作？模块导入正常但是没有 `index.db` 函数！

你需要 Biopython 1.57 或更新版本。（有 SQLite3 的 Python 支持）

29. `MultipleSeqAlignment` 对象在哪里？`Bio.Align` 模块导入正常但是这个类不在那里！

你需要 Biopython 1.54 或更新版本。或者，较早的 `Bio.Align.Generic.Alignment` 类支持它的一些功能，但是现在不推荐使用这个。

30. 为什么我不能直接从应用程序包装器上运行命令行工具？

你需要 Biopython 1.55 或更新版本。或者，直接使用 Python 的 `subprocess` 模块。

31. 我看到过一个代码的目录，但是我找不到那个能干嘛的代码了。它藏在哪儿了？

我们知道，我们的代码存放在 `__init__.py` 文件里。如果你此前没有在这个文件里寻找代码那么这可能会让人困惑。我们这样做的原因是为了让用户更容易导入。比如，不一定要像 `from Bio.GenBank import GenBank` 来导入一个“repetitive”，你仅需使用 `from Bio import GenBank` 就行。

32. 为什么 `CVS` 的代码貌似过期了？

2009 年 9 月下旬，在 Biopython 1.52 发布之后，我们从使用 `CVS` 转变为使用 `git`，`git` 是一个分散式的版本控制系统。旧的 `CVS` 服务仍可作为静态和只读备份，但是如果你想获取最新的代码，你需要使用 `git`。详见我们的网站获取更多信息。

对于更一般的问题，Python FAQ 页面 <http://www.python.org/doc/faq/> 可能会有帮助。

第 2 章快速开始——你能用 BIOPYTHON 做什么？

此部分旨在能让你快速开始 Biopython，并给你一个大概的了解什么可用以及如何使用它。此部分的所有例子都会假设你有 Python 的基础知识，并且前提是你已经在你的系统上安装了 Biopython。如果你认为你需要认真复习 Python，主流的 Python 网站提供了相当多的免费文档，你可以从以下网站开始 (<http://www.python.org/doc/>)。

由于计算机上大量的生物学工作涉及到网上的数据库，某些例子也会需要联网才能完成。

让我们看看我们能用 Biopython 做什么吧。

2.1 Biopython 功能概览

正如介绍中提到的，Biopython 是一个库的集合，这个库能在计算机上工作的生物学家解决感兴趣的事情。一般来说，你至少需要一点编程经验（当然是 Python！）或至少有兴趣学习编程。Biopython 的任务就是通过提供可重复利用的库，让编程人员的工作变得更加容易，可以使你能集中精力解决你所感兴趣的问题，而不用花太多精力去完成一个解析特殊文件格式的构件（当然，如果你想帮我们写一个原本不存在的解析器并把它贡献给 Biopython，请继续！）。所以 Biopython 的工作是让你更加轻松！

值得一提的是，Biopython 通常能给出多种方式来解决“相同的事情”。在最近的版本中，情况有所改善，但这仍可让人沮丧，因为在理想的 Python 中应该只有一种正确的方式去解决问题。但是，这也可以成为一个真正的好处，因为它给了你很多灵活性和对库的控制。本教程给你展示普通或简单的方式去处理问题以便于你能自己处理事情。想要学习更多替代的方法，请查看 Cookbook（第 18 章，这里有一些很酷的技巧和提示），进阶部分（第 20 章），内建“文档”（通过 Python help 命令），或者 API 文档）或者代码本身。

2.2 处理序列

生物信息学的中心对象是序列。因此，我们先快速开始介绍一下 Biopython 处理序列的机制，主要是 Seq 对象，这个我们也将会在第 3 章中详细讨论。

大多数时候当我们想到一条序列时，在我们脑海中都会有一串类似于‘AGTACTGGT’的字母串。你可以按以下步骤创建一个 Seq 对象——“>>>”表示 Python 提示符后紧跟你要输入的内容：

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACTGGT")
>>> my_seq
Seq('AGTACTGGT', Alphabet())
>>> print my_seq
AGTACTGGT
>>> my_seq.alphabet
Alphabet()
```


这里是一个由通用字母表组成的序列对象——说明我们还没有指定它是一条 DNA 还是蛋白质序列(好吧,一个有很多的 Ala, Gly, Cys 和 Thr 蛋白质序列!(幽默))。在第3章我们将讨论更多关于字母表。

除了有一个字母表, Seq 对象支持不同于 Python 的字符串方法。你不能对一个纯字符串做以下处理:

```
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.complement()
Seq('TCATGTGACCA', Alphabet())
>>> my_seq.reverse_complement()
Seq('ACCACTGTACT', Alphabet())
```

另一个最重要的类是 SeqRecord 或 Sequence Record。它保留了一条序列(作为 Seq 对象)额外的注释信息,包括 ID, name 和 description。用于读写序列文件格式的 Bio.SeqIO 模块能与 SeqRecord 对象一起工作,稍后我们将会介绍,详细内容在第5章。

这涵盖了基本的功能和 Biopython 序列类的使用。现在你应该有一些想法像怎么和 Biopython 库互动,是时候去钻研它的乐趣,探讨处理生物学文件格式的有趣世界了!

2.3 用法示例

在我们跳到语法解析器和 Biopython 处理的其他所有事之前,让我们先建立一个例子来激励我们所做的事情并让生活变得更加有趣。毕竟,如果没有任何生物学在这篇教程里,那你为什么还要读它呢?

因为我喜欢植物,我想我们就来一个基于植物的例子吧(对其他生物的爱好者对不起了!)。刚刚结束了我们当地的一个温室的旅程,我们对 Lady Slipper Orchids 突然有一个难以置信的迷恋(如果你想知道为什么,请看看一些 Lady Slipper Orchids 的照片,并试试 Google 图片搜索)。

当然,兰花不仅仅只有外观好看,它们也极力吸引着人们研究其进化和系统分类学。假设我们正在考虑写一份关于 Lady Slipper Orchids 进化研究的基金方案,我们就会想看看别人已经做了什么样的研究,然后我们能够增加一些什么内容。

经过一些阅读之后,我们发现 Lady Slipper Orchids 属于兰科拖鞋兰亚科并且由 5 个属组成: *Cypripedium*, *Paphiopedilum*, *Phragmipedium*, *Selenipedium* 和 *Mexipedium*。

这已经给了我们足够多的信息来探究更多的东西。现在,让我们看看 Biopython 工具能起到怎样的作用。我们从一条从 2.4 部分解析出来的序列开始,但是我们稍后还是回到兰花上来——比如我们将在 PubMed 上搜索有关兰花的文章然后在 GenBank 上提取序列(第9章),从 Swiss-Prot 上提取特定的兰花蛋白数据(第10章),最后在 6.4.1 部分我们用 ClustalW 对兰花蛋白进行多序列比对。

2.4 解析序列文件格式

很多生物信息学工作的一大部分都会涉及到处理各种包含有生物学数据的文件格式类型。这些文件保存了有趣的生物学数据,因而一个特殊的挑战是需要将这些文件解析成你能使用某种编程语言操作的格式。然而这些解析工作有时会让人感到失望,因为这些格式有可能经常改变,而一个细微的改变也有可能让设计得最好的解析器失去作用。

我们现在开始简单地介绍 Bio.SeqIO 模块——你可以在第5章中查看更多。我们从在线搜索我们的朋友——Lady Slipper Orchids——开始。为尽量保持简单,我们仅仅手动使用 NCBI 网站。我们先看看 NCBI 上的 nucleotide 库,使用在线的 Entrez 搜索(<http://www.ncbi.nlm.nih.gov:80/entrez/query.fcgi?db=Nucleotide>)包含 Cypripedioideae 所有东西(这是 Lady Slipper Orchids 的亚科)。

当本教程最初编写时，这个搜索仅给我们找到了 94 条匹配的信息，我们将结果保存为 FASTA 格式文本文件和 GenBank 格式文本文件（文件 `ls_orchid.fasta` 和 `ls_orchid.gbk`，也包含在 Biopython 源代码包下 `docs/tutorial/examples/`）。

如果你现在搜索，你将会获得几百个的匹配结果！跟着教程，如果你想要看看相同的基因列表，请下载上面两个文件或者从 Biopython 源代码中拷贝 `docs/examples/`。在 2.5 部分我们将会看到怎样使用 Python 做类似的搜索。

2.4.1 简单的 FASTA 解析示例

如果你用你喜好的文本编辑器打开了 lady slipper orchids 的 FASTA 文件 `ls_orchid.fasta`，你会看到文件开头像这样：

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTACCGGGGCGATTGCTCCCGTGGTGACCGTGATTTGTTGTTGGG
...
```

它包含有 94 条记录，每一行都以“>”开头，（大于号）紧随其后的是一行或多行序列。现在试试以下 Python 代码：

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
```

你应该会得到类似这样的一些东西出现在屏幕上：

```
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', SingleLetterAlphabet())
740
...
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', SingleLetterAlphabet())
592
```

注意 FASTA 文件并没有指定字母表，因此 `Bio.SeqIO` 默认使用相当通用的 `SingleLetterAlphabet()` 而不是 DNA 序列特有的。

2.4.2 简单的 GenBank 解析示例

现在我们来加载一个 GenBank 文件 `ls_orchid.gbk` ——注意这里的代码与上面处理 FASTA 文件的代码几乎完全相同——仅有的不同之处是我们改变了文件名和格式的字符串：

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
```

这段代码应该会给出：

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
...
```

```
Z78439.1
```

```
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
```

```
592
```

这一次 Bio.SeqIO 能够选择一个合理的字母表，IUPAC Ambiguous DNA。你应该注意到了这个例子中有一个较短的字符串被作为 seq_record.id。

2.4.3 我爱解析——请别停止讨论它！

Biopython 有很多的解析器，基于它们所解析的文件格式，每一个都有自己独特的作用。第5章包含 Bio.SeqIO 更详细的内容，而第6章将介绍用于序列比对的 Bio.AlignIO。

由于最主流的文件格式都有解析器整合在 Bio.SeqIO 和/或 Bio.AlignIO 中，对于一些比较罕见的或者不被人们喜爱的文件格式，要么根本就没有解析器，要么就是一些没有链接的老的解析器。请到 wiki 页面 <http://biopython.org/wiki/SeqIO> 以及 <http://biopython.org/wiki/AlignIO> 查看最新信息，或者咨询邮件列表。wiki 页面上应该包含了支持文件类型的最新列表，还有一些附加的例子。

另一个查找特定解析器信息和如何很酷的使用它们的地方就是 Cookbook (本教程的第18章)。如果你没有找到你要的信息，请考虑及时帮帮你那可悲的过劳的文档，并提交一份 cookbook entry！（一旦你知道怎么做了，那就是了！）

2.5 连接生物学数据库

在生物信息学中你需要做的很普遍的事情之一是从生物学数据库中提取信息。手动访问这些数据库可能会非常枯燥乏味，尤其是当你有很多重复的工作要做的时候。Biopython 试图通过用 Python 脚本访问一些可用的在线数据库来节省你的时间和精力。目前，Biopython 有从以下数据库中获取信息的代码：

- NCBI 的 Entrez (和 PubMed) ——见第9章。
- ExPASy ——见第10章。
- SCOP——见 Bio.SCOP.search() 方法。

使用模块里的代码基本上可以容易地写出与这些页面中 CGI 脚本交互的 Python 代码，因此你能很方便地获得想要的结果。在某些情况下，结果能很好地整合到 Biopython 解析器中从而使得提取信息更加简单。

2.6 下一步做什么

现在你已经做到这一步，你应该对基本的 Biopython 有一个很好的了解，并准备好开始用它完成一些有用的工作。现在最好先完成阅读本教程，然后如果你可能会想看看源码以及文档。

一旦你知道你想做什么，以及 Biopython 能完成它的库，你应该看看 Cookbook (第18章)，在这里可能会有一些类似你工作的示例代码。

如果你知道你想要做什么，但是还没弄明白怎么去做，请随时将你的问题贴出到主要的 Biopython 列表中（见 http://biopython.org/wiki/Mailing_lists）。这不仅方便我们回答你的问题，也有助于我们改进文档以便于它能帮到下一个和你做同样工作的人。

请享受代码吧！

第 3 章生物序列对象

生物学序列以绝对的优势成为生物信息学研究的重点对象。这一章我们将简要介绍 Biopython 处理这些序列的机制 – Seq 对象。第 4 章将要引入与此相关的 SeqRecord 对象，此对象可以将序列信息和注释结合起来，用于第 5 章序列的输入/输出。

序列实质上就是由字母构成的字符串，比如 AGTACACTGGT，看起来很自然，因为这就是序列在生物学文件中的常用代表格式。

Seq 对象和标准的 Python 字符串有两个明显的不同。首先，它们使用不同的方法。尽管“Seq”对象支持常规字符串的很多方法，但是它的 translate() 方法在做生物学翻译时是不同的。相似的还有其他的生物学相关的方法，比如 reverse_complement()。其次，Seq 对象具有一个重要的属性 – alphabet，这一对象用于描述由单个字母构成的序列字符串的“mean”（意义），以及如何解释这一字符串。例如，AGTACACTGGT 序列是个 DNA 序列还是一段富含 Alanines, Glycines, Cysteines and Threonines 的蛋白质序列？

3.1 序列和字母表

字母对象可能是使得“Seq”对象不仅仅是字符串的重要因素。目前，Biopython 的字母表定义在“Bio.Alphabet”模块。我们将会使用 IUPAC 字母表 (<http://www.chem.qmw.ac.uk/iupac/>) 来处理我们比较青睐的对象：DNA、RNA 和蛋白质对象。

Bio.Alphabet.IUPAC 提供了蛋白质、DNA 和 RNA 的基本定义，并且提供了扩展和定制基本定义的功能。例如蛋白质，有一个基本的 IUPACProtein 类，另外还有一个 ExtendedIUPACProtein 类。这个类包含除 20 种常见氨基酸外的其他氨基酸元素，比如“U”（或“Sec”代表硒代半胱氨酸），“O”（或“Pyl”代表吡咯赖氨酸），还有歧意字母“B”（或“Asx”代表天冬酰胺或者天冬氨酸），“Z”（或“Glx”代表谷氨酰胺或者谷氨酸），“J”（或“Xle”代表亮氨酸或异亮氨酸），“X”（或“Xxx”代表未知氨基酸）。同理，对于 DNA 有 IUPACUnambiguousDNA、IUPACAmbiguousDNA 和 ExtendedIUPACDNA 类，分别提供基本字母，每种可能下的歧意字母和修饰后的碱基。同样地，RNA 可以使用 IUPACAmbiguousRNA 和 IUPACUnambiguousRNA 类。

使用字母表类有两方面的优势。首先，明确了 Seq 对象包含的信息的类型；其次通过类型检查，它提供了约束信息的工具。

我们已经知道了将要处理的对象，现在让我们看看怎么使用这些类做一些有意思的事情。你可以创建一条有通用字母组成的模糊序列，如下：

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> my_seq.alphabet
Alphabet()
```

然而，如果可能，你要在创建序列对象的时候就尽量明确指定字母的类型，如下创建一条明确的 DNA 字母表对象。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

当然, 除非这真的是一个氨基酸序列:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

3.2 序列表现的像字符串一样

在许多时候, 我们可以讲 Seq 对象处理成正常的 Python 字符串, 比如取序列长度, 迭代元素:

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCG", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seq):
...     print index, letter
0 G
1 A
2 T
3 C
4 G
>>> print len(my_seq)
5
```

你可以像字符串那样获取序列的元素 (但是请记住, Python 计数从 0 开始):

```
>>> print my_seq[0] #first letter
G
>>> print my_seq[2] #third letter
T
>>> print my_seq[-1] #last letter
G
```

Seq 对象有一个 .count() 方法, 类似于字符串。记住这意味就像 Python 的字符串一样进行着非重叠的计数。

```
>>> from Bio.Seq import Seq
>>> "AAAA".count("AA")
2
>>> Seq("AAAA").count("AA")
2
```

但是在某些生物学上, 你可能需要使用重叠计数 (就像上面的例子中如果重复计数结果将为 3)。当计算耽搁字母出现的次数时, 重叠计数和非重叠计数没有差别。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAAATCGC', IUPAC.unambiguous_dna)
>>> len(my_seq)
32
>>> my_seq.count("G")
9
>>> 100 * float(my_seq.count("G") + my_seq.count("C")) / len(my_seq)
46.875
```

你当然可以使用上面的代码段计算 GC 含量，但是记住 Bio.SeqUtils 模块已经建立了好几个 GC 函数，类如：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqUtils import GC
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAAATCGC', IUPAC.unambiguous_dna)
>>> GC(my_seq)
46.875
```

注意在使用 Bio.SeqUtils.GC() 函数时会自动处理序列和可代表 G 或者 C 的歧义核苷酸字母 S 混合的情况。

然后还要注意，就像正常的 Python 字符串，Seq 对象在某些方式下是只读的。如果需要编辑序列，比如模拟点突变，请看后续的 3.12 章节中讲述的 MutableSeq 对象。

3.3 切取序列

一个较为复杂的例子，让我们切取序列。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq[4:12]
Seq('GATGGGCC', IUPACUnambiguousDNA())
```

要注意两个有意思的地方。首先，序列第一个元素从 0 开始，这是符合 Python 字符串的规则。这在计算机科学上是普遍现象，但在生物学上不是这样。当你做切片的时候，第一项包含了（比如例子中的 4），而最后一项去除了（例子中的 12）。这是 Python 的规则，但当然这不是世界上所有人都希望的。主要是为了和 Python 保持一致。

第二个需要注意的地方是，切片是在序列数据字符串上执行的，但是产生的新对象保留了原始 Seq 对象的字母表信息。

同样和 Python 字符串一样，你可以通过设置起始位置、终止位置和 步幅（间隔数，默认为 1）进行切片。例如，我们可以分别获取下面 DNA 序列密码子第一、第二、第三位的碱基。

```
>>> my_seq[0:3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
>>> my_seq[1:3]
Seq('AGGCATGCATC', IUPACUnambiguousDNA())
>>> my_seq[2:3]
Seq('TAGCTAAGAC', IUPACUnambiguousDNA())
```

你可能已经注意到 Python 字符串中的另一个奇特步幅设定：使用 -1 返回倒序字符串切片。当然以也可以使用 Seq 对象来完成。

```
>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

3.4 将序列对象转换成字符串

如果你仅仅需要一个单纯的字符串，就像写入文件或者插入数据库，这事很容易就可以实现的：

```
>>> str(my_seq)
'GATCGATGGGCCTATATAGGATCGAAAAATCGC'
```

尽管对 Seq 对象调用 str() 方法将以字符串的形式返回全长序列，但是你经常不需要特地做这个转换。当使用 print 打印声明是，Python 会自动转换。

```
>>> print my_seq
GATCGATGGGCCTATATAGGATCGAAAAATCGC
```

当你进行 Python 字符串格式化或者插入操作符 (%) 时，可以直接把 Seq 对象和 %s 占位符一起使用：

```
>>> fasta_format_string = ">Name\n%s\n" % my_seq
>>> print fasta_format_string
>Name
GATCGATGGGCCTATATAGGATCGAAAAATCGC
```

这一行代码展示的是一个简单的 FASTA 格式的记录（不用关心自动换行）。4.5 部分将介绍一个简洁的方式从 SeqRecord 对象中获取 FASTA 格式的字符串，更详细的读写 FASTA 格式的序列文件将在第 5 章介绍。

注意：如果你使用 Biopython 1.44 或者更旧的版本，使用 str(my_seq) 只会返回一个截短了的序列。这时候可以使用 my_seq.tostring()，为了保持向后兼容性，这一方法在当前的 Biopython 版本中还有保留。

```
>>> my_seq.tostring()
'GATCGATGGGCCTATATAGGATCGAAAAATCGC'
```

3.5 连接或添加序列

当然，原则上你可以将任何两个 Seq 对象加在一起，就像 Python 字符串一样去连接它们。但是你不能将两个不相容的字母表加在一起，比如蛋白质序列和核苷酸序列就不能简单叠加。

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq
Traceback (most recent call last):
...
TypeError: Incompatible alphabets IUPACProtein() and IUPACUnambiguousDNA()
```

如果你真的想这么做，你必须首先将两个序列转换成通用字母表。

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
```

```
>>> protein_seq + dna_seq
Seq('EVRNAKACGT', Alphabet())
```

这里有个例子是将通用核苷酸序列加到明确的 IUPAC DNA 序列上，最后生成一段模糊的核苷酸序列。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_nucleotide
>>> from Bio.Alphabet import IUPAC
>>> nuc_seq = Seq("GATCGATGC", generic_nucleotide)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> nuc_seq
Seq('GATCGATGC', NucleotideAlphabet())
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> nuc_seq + dna_seq
Seq('GATCGATGCACGT', NucleotideAlphabet())
```

3.6 改变大小写

Python 字符串具有很有用的转换大小写的 `upper` 和 `lower` 方法。从 Biopython 1.53 起，`Seq` 对象也获取了类似的方法应用于字母表。例如：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> dna_seq = Seq("acgtACGT", generic_dna)
>>> dna_seq
Seq('acgtACGT', DNAAlphabet())
>>> dna_seq.upper()
Seq('ACGTACGT', DNAAlphabet())
>>> dna_seq.lower()
Seq('acgtacgt', DNAAlphabet())
```

这在不区分大小写进行匹配的时候很有用。

```
>>> "GTAC" in dna_seq
False
>>> "GTAC" in dna_seq.upper()
True
```

注意，严格地说 IUPAC 字母表仅仅是对于大写字母构成的序列的，因此：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> dna_seq
Seq('ACGT', IUPACUnambiguousDNA())
>>> dna_seq.lower()
Seq('acgt', DNAAlphabet())
```

3.7 核苷酸序列和（反向）互补序列

对于核苷酸序列，你可以使用 `Seq` 对象内置的方法很容易地获得 `Seq` 的互补或反向互补序列。


```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAAATCGC', IUPACUnambiguousDNA())
>>> my_seq.complement()
Seq('CTAGCTACCCGATATATCCTAGCTTTTAGCG', IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()
Seq('GCGATTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
```

在前面的方法中，使用切片的 -1 的步长可以很容易的获取一个 Seq 对象的反向序列。

```
>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

在所有这些操作中，字母的属性一直保留着。这是非常有用的，以防你不小心做一些奇怪的事情，比如获取蛋白质序列的（反向）互补序列。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> protein_seq.complement()
Traceback (most recent call last):
...
ValueError: Proteins do not have complements!
```

:ref:`5.5.3 <sec-SeqIO-reverse-complement>` 部分的例子将 ``Seq`` 对象的反向互补方法和 ``Bio.SeqIO`` 对于序列的输入/输出方法结合起来。

3.8 转录

在谈论转录之前，我想先说明一下链的问题。考虑以下（编造的）编码短肽的双链 DNA 的延伸：

DNA coding strand (aka Crick strand, strand+1)

5' ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG

3'

|||||

3' TACCGGTAACATTACCGGCGACTTTCCACGGGCTATC

5'

DNA template strand (aka Watson strand, strand-1)

|

Transcription

↓

5' ATGGCCCGATAG
3' |||||
3' CACGGGCTATC
5'

Transcription
↓
5' AUGCCCGAUAG
3'

实际的生物学上的转录过程是将模板链反向互补 (TCAG → CUGA) 生成 mRNA。但是，在 Biopython 和 生物信息学领域，我们通常会直接利用编码链，因为我们可以通过 T → U 的转换获得 mRNA。

现在让我们着手真实地使用 Biopython 做一个转录。首先，让我们分别创建 DNA 序列的编码链和模板链的 Seq 对象：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> template_dna = coding_dna.reverse_complement()
>>> template_dna
Seq('CTATCGGGCACCCCTTTCAGCGGCCATTACAATGGCCAT', IUPACUnambiguousDNA())
```

这是和上面的图表相一致的，记住按照惯例核苷酸序列通常是从 5'到 3'端方向的，而图中所示的模板链是反向的。

现在让我们使用 Seq 对象内置的 transcribe 方法将编码链转录成对应的 mRNA：

```
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

就如你看到的，这里做的全部工作是将 $T \rightarrow U$ 转换，并调整字母表。

如果你确实想从模板链去做一个真正的生物学上的转录，需要两步：

```
>>> template_dna.reverse_complement().transcribe()
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Seq 对象还包含了从 mRNA 逆向转录为 DNA 编码链的方法。同样，这仅仅是从 $U \rightarrow T$ 的替代并伴随着字母表的变化：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.back_transcribe()
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
```

注意：Seq 对象的 transcribe 和 back.transcribe 方法直到 Biopython 1.49 版本才出现，在较早的版本中你需要使用 Bio.Seq 模块的函数替代，详见 3.14 部分。

3.9 翻译

继续使用在转录那个小节中的例子，现在让我们将这个 mRNA 翻译成相对应的蛋白质序列，利用的是 Seq 对象众多生物学方法中的一个：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG", IUPAC.unambiguous_rna)
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
>>> messenger_rna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

你也可以直接从编码 DNA 链进行翻译：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

你应该注意到在上面的蛋白质序列中，除了末尾的终止符外，在序列中间还有一个终止符。其实这是一个精心选择的例子，因为由它我们可以引申讲一下可选参数，包括不同的翻译表（遗传密码）。

Biopython 上可用的翻译表是基于 NCBI（参考这个教程的下一个部分）。默认情况下，翻译使用的是标准遗传密码（NCBI 上 table id 1）。假设我们需要翻译一个线粒体序列，我们就需要告诉翻译函数使用相关的遗传密码：

```
>>> coding_dna.translate(table="Vertebrate Mitochondrial")
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

你也可以利用 NCBI 上表格的标号来指定所使用的遗传密码，这样更简洁一些，在 GenBank 文件的特征注释中经常包含表格的标号：

```
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
```

现在你可能想将上面的核苷酸序列仅翻译到阅读框的第一个终止密码子，然后停止（这更符合自然现象）。

```
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
>>> coding_dna.translate(to_stop=True)
Seq('MAIVMGR', IUPACProtein())
>>> coding_dna.translate(table=2)
Seq('MAIVMGRWKGAR*', HasStopCodon(IUPACProtein(), '*'))
>>> coding_dna.translate(table=2, to_stop=True)
Seq('MAIVMGRWKGAR', IUPACProtein())
```

注意到当你使用 `to_stop` 参数时，终止密码子本身是不翻译的，终止的符号也是不显现在蛋白质序列中的。

如果你不喜欢默认的星号作为终止符号，你也可以自己指定终止符。

```
>>> coding_dna.translate(table=2, stop_symbol="@")
Seq('MAIVMGRWKGAR@', HasStopCodon(IUPACProtein(), '@'))
```

现在假设你有一条完整的编码序列 CDS，这是一种核苷酸序列（例如 mRNA 剪切以后），序列全长都是密码子（也就是长度是 3 的倍数），开始于起始密码子，终止于终止密码子，阅读框内没有内部的终止密码子。通常情况下，给你一条完整的 CDS，默认的翻译方法即可以翻译出你想要的（有时使用 `to_stop` 选项）。但是，如果序列使用的是非标准的起始密码子呢？这种情况在细菌中很常见，比如 *E. coli* K12 中的基因 *yaaX*：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> gene = Seq("GTGAAAAAGATGCAATCTATCGTACTCGCACTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \
...            "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAATTACAGATAGGCGATCGTGAT" + \
...            "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCCGCGTGGTGGAAACAACAT" + \
...            "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCGCGCCACCAT" + \
...            "AAGAAAGCTCCTCATGATCATCAGGCGGTCTATGGTCCAGGCAAACATCACCGCTAA",
...            generic_dna)
>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR*',
HasStopCodon(ExtendedIUPACProtein(), '*'))
>>> gene.translate(table="Bacterial", to_stop=True)
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

在细菌遗传密码中 GTG 是个有效的起始密码子。正常情况下 编码缬氨酸，如果作为起始密码子，则翻译成甲硫氨酸。当你告诉 Biopython 你的序列是完整 CDS 时，这事将会发生。

```
>>> gene.translate(table="Bacterial", cds=True)
Seq('MKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

除了告诉 Biopython 翻译时使用另一种起始密码子编码甲硫氨酸外，使用这一选项同样能确保你的序列是个真实有效的 CDS（如果不是将会抛出异常）。

第18.1.3章的例子将把 Seq 对象的翻译方法和 Bio.SeqIO 对象的对于序列的输入/输出方法结合起来。

3.10 翻译表

在前面的章节中我们讨论了 Seq 对象的转录方法 (并且提到了 Bio.Seq 模块中的等效函数—参见第3.14章节)。实质上使用的这些密码子表对象来自与 NCBI 的 `ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt` `<ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt>‘_` , 还有 `http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi` 以一种更易读的形式呈现。

和前面一样, 让我们仅仅关注两个选择: 标准的翻译表和脊椎动物线粒体 DNA 的翻译表。

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> mito_table = CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
```

另一种方式, 这些表也可以分别以标号 1 和 2 来标识:

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_id[1]
>>> mito_table = CodonTable.unambiguous_dna_by_id[2]
```

你可以在打印后直观地比较这些实际的翻译表:

```
>>> print standard_table
Table 1 Standard, SGC0
```

	T	C	A	G	
T	TTT F	TCT S	TAT Y	TGT C	T
T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA Stop	A
T	TTG L(s)	TCG S	TAG Stop	TGG W	G
C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L(s)	CCG P	CAG Q	CGG R	G
A	ATT I	ACT T	AAT N	AGT S	T
A	ATC I	ACC T	AAC N	AGC S	C
A	ATA I	ACA T	AAA K	AGA R	A
A	ATG M(s)	ACG T	AAG K	AGG R	G
G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V	GCG A	GAG E	GGG G	G

和

```
>>> print mito_table
Table 2 Vertebrate Mitochondrial, SGC1
```

	T	C	A	G	

T	TTT F	TCT S	TAT Y	TGT C	T
T	TTC F	TCC S	TAC Y	TGC C	C
T	TTA L	TCA S	TAA Stop	TGA W	A
T	TTG L	TCG S	TAG Stop	TGG W	G

C	CTT L	CCT P	CAT H	CGT R	T
C	CTC L	CCC P	CAC H	CGC R	C
C	CTA L	CCA P	CAA Q	CGA R	A
C	CTG L	CCG P	CAG Q	CGG R	G

A	ATT I(s)	ACT T	AAT N	AGT S	T
A	ATC I(s)	ACC T	AAC N	AGC S	C
A	ATA M(s)	ACA T	AAA K	AGA Stop	A
A	ATG M(s)	ACG T	AAG K	AGG Stop	G

G	GTT V	GCT A	GAT D	GGT G	T
G	GTC V	GCC A	GAC D	GGC G	C
G	GTA V	GCA A	GAA E	GGA G	A
G	GTG V(s)	GCG A	GAG E	GGG G	G

你会发现下面的特性很有用，比如当你查找新基因时：

```
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.forward_table["ACG"]
'T'
```

3.11 比较 Seq 对象

序列之间的比较实际上是一个比较复杂的话题，没有简单的方法来判断两个序列是等同的。核心的问题是字母的意义是依赖于上下文的。字母“A”既可以是 DNA、RNA 也可以使蛋白质序列的一部分。Biopython 在 Seq 对象中包含了字母表对象，以此尝试获得这些信息。所以比较两个 Seq 对象意味着既要考虑两个序列的字符串 又要 考虑字母表。

举个例子，你可能会觉得 Seq("ACGT", IUPAC.unambiguous_dna) 和 Seq("ACGT", IUPAC.ambiguous_dna) 这两个 DNA Seq 对象是一样的，尽管它们确实具有不同的字母表。根据上下文来判断是很重要的。

下面这种情况更遭：假设你认为 Seq("ACGT", IUPAC.unambiguous_dna) 和 Seq("ACGT")（也就是默认的通用字母表）是等同的。那么依照逻辑，Seq("ACGT", IUPAC.protein) 和 Seq("ACGT") 也是等同的。现在从理论上讲，如果 $A=B$ ， $B=C$ ，那么通过递延性，我们会期望 $A=C$ 。因此遵从逻辑上的一致性我们需要将 Seq("ACGT", IUPAC.unambiguous_dna) 和 Seq("ACGT", IUPAC.protein) 等同起来，虽然大部分人会同这一递延，但是这是错误的。这一递延性的问题也会影响使用 Seq 对象作为 Python 字典的键值。

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.unambiguous_dna)
```

那么接下来 Biopython 会怎么做？等同性测试是 Python 对象默认要做的测试。经过检验查看内存中的对象是不是同一个。这是一个非常严格的测试：

```
>>> seq1 == seq2
False
>>> seq1 == seq1
True
```

如果你真想这么做，你可以更明确地使用 Python 中的 `id` 函数，

```
>>> id(seq1) == id(seq2)
False
>>> id(seq1) == id(seq1)
True
```

在日常使用中，你的所有序列可能都是同一个字母表，或者至少都是同一类型的序列（都是 DNA、RNA 或者都是蛋白质）。你可能想要的只是以字符串的形式比较这些序列，那么直接这么做：

```
>>> str(seq1) == str(seq2)
True
>>> str(seq1) == str(seq1)
True
```

作为一个扩展，你可以建立一个 Python 字典，以 Seq 对象作为键值。一般情况下，将序列作为字符串赋予键值更有用。详见 3.4 部分。

3.12 MutableSeq 对象

就像正常的 Python 字符串，Seq 对象是“只读的”，在 Python 术语上就是不可变的。除了想要 Seq 对象表现得向一个字符串之外，这是一个很有用的默认，因为在生物学应用上你往往需要确保你没有改动你的序列数据：

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

当你尝试编辑序列是你看看会发生什么：

```
>>> my_seq[5] = "G"
Traceback (most recent call last):
...
TypeError: 'Seq' object does not support item assignment
```

但是你可以使用 MutableSeq 对象将它转换成可变的序列，然后做任何你想要做的。

```
>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
```

或者你可以直接从字符串建立一个 MutableSeq 对象：

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)
```

这两种方式都可以将序列对象转换成可变的：

```
>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq[5] = "C"
>>> mutable_seq
```

```
MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.remove("T")
>>> mutable_seq
MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> mutable_seq
MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

请注意与 Seq 对象不同的是，MutableSeq 对象的各种方法都是实时呈现的，比如 reverse_complement() 和 reverse() 方法！

Python 中可变对象和不可变对象的一个重要的技术差别就是 MutableSeq 对象不可以作为字典的键值，但是 Python 字符串或者 Seq 对象就可以。

一旦你的 MutableSeq 对象编辑完成，很容易将它变回到只读的 Seq 对象，你只需：

```
>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

就像你从 Seq 对象中获取字符串一样，你也可以从 MutableSeq 获得（参见 3.4 章节）。

3.13 UnknownSeq 对象

UnknownSeq 对象是基本的 Seq 对象中的一个子类，其目的是一个已知长度的序列，但序列并不是由实际的字母组成的。在这种情况下，你当然可以将其作为一个正常的 Seq 对象，但是存储由一百万个“N”字母组成的字符串会浪费相当大量的内存，这时你可以只存储一个“N”和序列所需的长度（整数）。

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, alphabet = Alphabet(), character = '?')
>>> print unk
????????????????????
>>> len(unk)
20
```

当然你也可以指定一个字母，而不仅仅是“?”。一般核苷酸序列默认为“N”，蛋白质序列默认为“X”。

```
>>> from Bio.Seq import UnknownSeq
>>> from Bio.Alphabet import IUPAC
>>> unk_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous_dna)
>>> unk_dna
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> print unk_dna
NNNNNNNNNNNNNNNNNNNN
```

你可以使用所有常规的 Seq 对象，记住这些可以节省内存的 UnknownSeq 对象，如你所希望的那样在恰当的地方使用。

```
>>> unk_dna
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> unk_dna.complement()
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> unk_dna.reverse_complement()
UnknownSeq(20, alphabet = IUPACAmbiguousDNA(), character = 'N')
>>> unk_dna.transcribe()
```

```
UnknownSeq(20, alphabet = IUPACAmbiguousRNA(), character = 'N')
>>> unk_protein = unk_dna.translate()
>>> unk_protein
UnknownSeq(6, alphabet = ProteinAlphabet(), character = 'X')
>>> print unk_protein
XXXXXX
>>> len(unk_protein)
6
```

你也许能够在自己的代码中找到 `UnknownSeq` 对象的应用，但你更可能首先在由 `Bio.SeqIO` 创建的 `SeqRecord` 对象中遇到 `UnknownSeq` 对象（参见第5章）。一些序列格式的文件不总是由实际的序列组成，像 GenBank 和 EMBL 文件就可能包含各种特征的列表，而序列部分仅展示 contig 信息。又或者在测序工作中的 QUAL 文件仅包含质量分数，而从未包含序列，取而代之的和 QUAL 文件同时生成的 FASTA 格式文件确实是由序列构成。

3.14 直接使用字符串

在这一章的结尾，对于那些真的不想使用序列对象的人（或者那些更喜欢面向对象的函数式编程风格的人），`Bio.Seq` 的模块级别的函数可以接受普通的 Python 字符串，比如 `Seq` 对象（包括 `UnknownSeq` 对象）或者 `MutableSeq` 对象：

```
>>> from Bio.Seq import reverse_complement, transcribe, back_transcribe, translate
>>> my_string = "GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG"
>>> reverse_complement(my_string)
'CTAACCAGCAGCAGCACCCTTCCAACGACCCATAACAGC'
>>> transcribe(my_string)
'GCUGUUAUGGGUCGUUGGAAGGGUGGUCGUGCUGGUUAG'
>>> back_transcribe(my_string)
'GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGTTAG'
>>> translate(my_string)
'AVMGRWKGGRAAG*'
```

尽管如此，我们鼓励你使用默认的 `Seq` 对象。

第 4 章序列注释对象

第3章介绍了序列对象的基本情况。紧接上章的 Seq 类，这章主要讲 Sequence record 或称之为 SeqRecord 类，该类在 Bio.SeqRecord 模块中有定义。它（见 SeqFeature 对象）可使序列与高级属性（如 identifiers 和 features）关联。其应用贯穿序列输入/输出的交互界面 Bio.SeqIO 过程中（详见第5章）。

如读者只需处理 FASTA 格式的序列文件等简单数据，可略过本章。如涉及带注释内容的数据（如 GenBank 或 EMBL 格式文件），本章内容则非常重要。

尽管本章内容涵盖了 SeqRecord 和 SeqFeature 对象的大部分内容，但如需了解更多，读者可自行查阅 SeqRecord wiki (<http://biopython.org/wiki/SeqRecord>)，和内置帮助文档（或在线文档 SeqRecord 和 SeqFeature），获取更多信息：

```
>>> from Bio.SeqRecord import SeqRecord
>>> help(SeqRecord)
...
```

4.1 SeqRecord 对象

SeqRecord (Sequence Record) 类包含在 Bio.SeqRecord 模块中。该类是 Bio.SeqIO 序列输入/输出交互界面（详见第5章）的基本数据类型。可以把 identifiers 和 features 等高级属性与序列关联起来（参见第3章）。

SeqRecord 类非常简单，包括下列属性：

.seq –序列自身（即 Seq 对象）。

.id –序列主 ID（-字符串类型）。通常类同于 accession number。

.name –序列名/id（-字符串类型）。可以是 accession number，也可能是 clone 名（类似 GenBank record 中的 LOCUS id）。

.description –序列描述（-字符串类型）。

.letter_annotations –对照序列的每个字母逐字注释（per-letter-annotations），以信息名为键（keys），信息内容为值（value）所构成的字典。值与序列等长，用 Python 列表、元组或字符串表示。**.letter_annotations** 可用于质量分数（如第18.1.6节）或二级结构信息（如 Stockholm/PFAM 比对文件）等数据的存储。

.annotations –用于储存附加信息的字典。信息名为键（keys），信息内容为值（value）。用于保存序列的零散信息（如 unstructured information）。

.features –SeqFeature 对象列表，储存序列的结构化信息（structured information），如：基因位置，蛋白结构域。features 详见本章第三节（第4.3节）。

.dbxrefs –储存数据库交叉引用信息（cross-references）的字符串列表。

4.2 创建 SeqRecord

使用 SeqRecord 对象非常简单，因为所有的信息都存储在该类的属性中；通常不必手动新建，用 Bio.SeqIO 从序列文件读取即可（见第5章）。当然新建 SeqRecord 也不复杂。

4.2.1 从头新建 SeqRecord

SeqRecord 最少只需包含 Seq 对象：

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq)
```

还可以通过初始化函数给 id, name 和 description 赋值；反之，它们被设为默认值“unknown”(可随后编辑)：

```
>>> simple_seq_r.id
'<unknown id>'
>>> simple_seq_r.id = "AC12345"
>>> simple_seq_r.description = "Made up sequence I wish I could write a paper about"
>>> print simple_seq_r.description
Made up sequence I wish I could write a paper about
>>> simple_seq_r.seq
Seq('GATC', Alphabet())
```

标识符对输出 SeqRecord 内容到文件很重要，可随 SeqRecord 同时建立：

```
>>> from Bio.Seq import Seq
>>> simple_seq = Seq("GATC")
>>> from Bio.SeqRecord import SeqRecord
>>> simple_seq_r = SeqRecord(simple_seq, id="AC12345")
```

上述章节已提到，SeqRecord 含有一个 annotations 属性，用于储存各种杂乱注释的字典。添加 annotations 示例如下：

```
>>> simple_seq_r.annotations["evidence"] = "None. I just made it up."
>>> print simple_seq_r.annotations
{'evidence': 'None. I just made it up.'}
>>> print simple_seq_r.annotations["evidence"]
None. I just made it up.
```

letter.annotations 也是字典，其值为与序列等长的内置 Python 字符串、列表或元组：

```
>>> simple_seq_r.letter_annotations["phred_quality"] = [40,40,38,30]
>>> print simple_seq_r.letter_annotations
{'phred_quality': [40, 40, 38, 30]}
>>> print simple_seq_r.letter_annotations["phred_quality"]
[40, 40, 38, 30]
```

dbxrefs 和 features 分别是字符串和 SeqFeature 对象的 Python 列表，将在后续章节讨论。

4.2.2 根据 FASTA 文件创建 SeqRecord 对象

本节以鼠疫耶尔森菌株 (*Yersinia pestis* biovar *Microtus* str. 91001) 的 pPCP1 质粒全长序列为例，说明从 FASTA 文件创建 SeqRecord 的过程。该序列原始文件来自 NCBI，可在 Biopython 单元测试 GenBank

文件夹下找到，也可点击 [NC_005816.fna](#) 下载。

序列以大于号开头，该文件只包含一条序列：

```
>gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... pPCP1, complete sequence
TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGTAATCTGCTCTCC
...
```

回顾第2章的内容，我们已经遇到过 `Bio.SeqIO.parse(...)` 函数，用于遍历 `SeqRecord` 对象中的所有记录。此处，我们介绍 `Bio.SeqIO` 模块中的另一个类似函数 `Bio.SeqIO.read()`，用于读取单条序列的文件（详见第5章）：

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
SingleLetterAlphabet()), id='gi|45478711|ref|NC_005816.1|', name='gi|45478711|ref|NC_005816.1|',
description='gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... sequence',
dbxrefs=[])
```

现在让我们逐个介绍 `SeqRecord` 对象中的主要属性，从给予我们序列属性的 `Seq` 对象开始：

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG', SingleLetterAlphabet())
```

此处 `Bio.SeqIO` 默认为通用字母表（generic alphabet），而非判断是否 DNA 序列。如果 FASTA 文件中序列类型已知，也可通过 `Bio.SeqIO` 自行设定（见第5章用法）。

接下来介绍 `identifiers` 和 `description`：

```
>>> record.id
'gi|45478711|ref|NC_005816.1|'
>>> record.name
'gi|45478711|ref|NC_005816.1|'
>>> record.description
'gi|45478711|ref|NC_005816.1| Yersinia pestis biovar Microtus ... pPCP1, complete sequence'
```

FASTA 文件中序列名所在行的第一个单词（去除大于号后）被当作 `id` 和 `name`；而将整行（去除大于号后）作为 `description`。这样设定是为了向后兼容，同时也为了便于处理如下序列：

```
>Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1
TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGGGGTAATCTGCTCTCC
...
```

Note: 读取 FASTA 文件时其他注释属性为空：

```
>>> record.dbxrefs
[]
>>> record.annotations
{}
>>> record.letter_annotations
{}
>>> record.features
[]
```

本例中 FASTA 文件源于 NCBI，其规范的格式，意味着我们可以方便的解析这些信息并选择提取 `GI` 和 `accession number` 等信息。然后，对于从其他来源获得的 FASTA 文件，并不能确保能获得这些信息。

4.2.3 从 GenBank 文件创建 SeqRecord

仍以疫耶尔森菌株 pPCP1 质粒全长序列 (*Yersinia pestis* biovar *Microtus* str. 91001 plasmid pPCP1) 为例, 不同的是这次使用 Genbank 格式的文件, 该文件同样包含在 Biopython 单元测试/GenBank 文件夹下, 也可点击 [NC_005816.gb](#) 下载。

该文件只含一条记录 (只有一个 LOCUS 行):

```
LOCUS      NC_005816                9609 bp    DNA     circular BCT 21-JUL-2008
DEFINITION Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete
           sequence.
ACCESSION  NC_005816
VERSION    NC_005816.1  GI:45478711
PROJECT    GenomeProject:10638
...
```

同样使用 Bio.SeqIO 读取文件, 代码跟处理 FASTA 文件类似 (详见第5章):

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])
```

你可能已经发现了一些不同之处, 逐个环顾各个属性, 序列字符串和上述类似, 但此处 Bio.SeqIO 可自动识别序列类型 (详见第5章):

```
>>> record.seq
Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG', IUPACAmbiguousDNA())
```

name 源于 LOCUS 行, id 附加了版本后缀。description 源于 DEFINITION 行:

```
>>> record.id
'NC_005816.1'
>>> record.name
'NC_005816'
>>> record.description
'Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.'
```

GenBank 文件中 per-letter annotations 为空:

```
>>> record.letter_annotations
{}
```

多数注释信息储存在 annotations 字典中, 例如:

```
>>> len(record.annotations)
11
>>> record.annotations["source"]
'Yersinia pestis biovar Microtus str. 91001'
```

dbxrefs 列表中的数据来自 PROJECT 或 DBLINK 行:

```
>>> record.dbxrefs
['Project:10638']
```

最后也许也可能是最有意思的, features 列表以 SeqFeature 对象的形式保存了 features table 中的所有 entries (如 genes 和 CDS 等)。

```
>>> len(record.features)
29
```

接下来，我们将在第4.3节介绍 SeqFeature 对象。

4.3 Feature, location 和 position 对象

4.3.1 SeqFeature 对象

序列特征是描述一条序列不可或缺的部分。抛开序列本身，你需要一种方式去组织和获取关于这条序列的“抽象”信息。尽管设计一个通用的类囊括序列的所有特征看似是不可能的，但是 Biopython 的 SeqFeature 类试图尽可能多的囊括序列的所有特征。Biopython 主要依据 GenBank/EMBL 特征表来设计相应的对象，认识到这一点，将有助于你更快更好的理解 Biopython SeqFeature 对象。

SeqFeature 对象的关键目的在于描述其相对于父序列（parent sequence，通常为 SeqRecord 对象）所处的位置（location），通常是介于两个 positions 间的一个区域（region），后续第4.3.2节将详细说明。

SeqFeature 对象含大量属性，首先一一例出，然后在后续章节举例说明其用法：

.type –用文字描述的 feature 类型（如‘CDS’或‘gene’）。

.location – SeqFeature 在序列中所处的位置。见第4.3.2节。SeqFeature 设计了众多针对 location 对象的功能，包含一系列简写的属性。

.ref – .location.ref 简写 –location 对象相关的参考序列。通常为 None。

.ref_db – .location.ref_db 简写 – 指定 .ref 相关数据库名称。通常为 None。

.strand – .location.strand 简写 – 表示 feature 所处序列的 strand。在双链核酸序列中，1 表示正链，-1 表示负链，0 表示 strand 信息很重要但未知，None 表示 strand 信息未知且不重要。蛋白和单链核酸序列为 None。

.qualifiers –存储 feature 附加信息（Python 字典）。键（key）为值（value）所存信息的单字简要描述，值为实际信息。比如，键为“evidence”，而值为“computational (non-experimental)”。这只是为了提醒人们注意，该 feature 没有被实验所证实（湿实验）。Note：为与 GenBank/EMBL 文件中的 feature tables 对应，规定.qualifiers 中值为字符串数组（即使只有一个字符串）。

.sub_features –只有在描述复杂位置时才使用，如 GenBank/EMBL 文件中的‘joins’位置。已被 CompoundLocation 对象取代，因此略过不提。

4.3.2 Positions 和 locations

SeqFeature 对象主要用于描述相对于父序列中的位置（region）信息。Region 用 location 对象表示，通常是两个 position 间的范围。为了区分 location 和 position，我们定义如下：

position –表示位于序列中的单一位置，可以是精确的也可以是不确定的位置（如 5, 20, <100 和 >200）。

location –介于两个 positions 间的区域。比如 5..20（5 到 20）。

之所以特意提及这两个概念是因为我经常混淆两者。

4.3.2.1 FeatureLocation 对象

多数 SeqFeature 特别简单（真核基因例外），只需起点、终点以及 strand 信息。最基本的 FeatureLocation 对象中通常包括上述三点信息。

但实际情况未必如此简单，因为我们还需处理包含几个区域的复合 locations，而且 position 本身很可能是不精确的。

4.3.2.2 CompoundLocation 对象

为了更方便的处理 EMBL/GenBank 文件中的‘join’locations，Biopython 1.62 引入 CompoundLocation 对象。

4.3.2.3 模糊 Positions

目前，我们只处理过简单 position，feature location 复杂因素之一就是由 position 本身不准确所致。生物学中许多问题都是不确定的，比如：你通过双核苷酸 priming 证明了 mRNA 的转录起始位点是这两个位点中的一个。这是十分有价值的发现，但困难来自于怎样表述这个位点信息。为了处理类似情况，我们用模糊位点 (fuzzy position) 表示。根据 fuzzy position 的不同，我们用 5 个类分别描述：

ExactPosition –精确位点，用一个数字表示。从该对象的 position 属性可得知精确位点信息。

BeforePosition –位于某个特定位点前。如 ‘<13’，在 GenBank/EMBL 中代表实际位点位于 13 之前。从该对象的 position 属性可得知上边界信息。

AfterPosition –与 BeforePosition 相反，如 ‘>13’，在 GenBank/EMBL 中代表实际位点位于 13 以后。从该对象的 position 属性可获知下边界信息。

WithinPosition –介于两个特定位点之间，偶尔在 GenBank/EMBL locations 用到。如 ‘(1.5)’，GenBank/EMBL 中代表实际位点位于 1 到 5 之间。该对象需要两个 position 属性表示，第一个 position 表示下边界 (本例为 1)，extension 表示上边界与下边界的差值 (本例为 4)。因此在 WithinPosition 中，object.position 表示下边界，object.position + object.extension 表示上边界。

OneOfPosition –表示几个位点中的一个 (GenBank/EMBL 文件中偶尔能看到)，比如在基因起始位点不明确或者有两个候选位点的时候可以使用，或者用于明确表示两个相关基因特征时使用。

UnknownPosition –代表未知位点。在 GenBank/EMBL 文件中没有使用，对应 UniProt 中的 ‘?’ feature 坐标。

举例说明创建一个 fuzzy end points:

```
>>> from Bio import SeqFeature
>>> start_pos = SeqFeature.AfterPosition(5)
>>> end_pos = SeqFeature.BetweenPosition(9, left=8, right=9)
>>> my_location = SeqFeature.FeatureLocation(start_pos, end_pos)
```

Note : Biopython 1.59 以后，fuzzy-locations 有修改，特别是 BetweenPosition 和 WithinPosition，现在必须显示用整数表示。起点为较小值，终点则为较大值。

print 输出 FeatureLocation 对象，可看到简洁的结果:

```
>>> print my_location
[>5:(8^9)]
```

也可通过 start 和 end 属性得到 fuzzy position 的起始/终止位点:

```
>>> my_location.start
AfterPosition(5)
>>> print my_location.start
>5
>>> my_location.end
BetweenPosition(9, left=8, right=9)
>>> print my_location.end
(8^9)
```


如果你只想获取数字，不理睬模糊 positions，则可将 fuzzy position 强制转换成一个整数：

```
>>> int(my_location.start)
5
>>> int(my_location.end)
9
```

为了兼容旧版 Biopython，保留了整数形式的 `nofuzzy_start` and `nofuzzy_end`：

```
>>> my_location.nofuzzy_start
5
>>> my_location.nofuzzy_end
9
```

Notice：上述例子只是为了帮助你理解 fuzzy locations。

相似的，如果要建立一个精确 location，只需将整数传递给 `FeaturePosition` 构造函数，即可建立 `ExactPosition` 对象：

```
>>> exact_location = SeqFeature.FeatureLocation(5, 9)
>>> print exact_location
[5:9]
>>> exact_location.start
ExactPosition(5)
>>> int(exact_location.start)
5
>>> exact_location.nofuzzy_start
5
```

以上是 Biopython 处理 fuzzy position 的实现方法。希望读者能体会之所以这样设计，都是为了使用上的方便（至少不比精确位点复杂）

4.3.2.4 Location testing

可用 Python 关键词 `in` 检验某个碱基或氨基酸残基的父坐标是否位于 feature/location 中。

假定你想知道某个 SNP 位于哪个 feature 里，并知道该 SNP 的索引位置是 4350 (Python 计数)。一个简单的实现方案是用循环遍历所有 features：

```
>>> from Bio import SeqIO
>>> my_snp = 4350
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> for feature in record.features:
...     if my_snp in feature:
...         print feature.type, feature.qualifiers.get('db_xref')
...
source ['taxon:229193']
gene ['GeneID:2767712']
CDS ['GI:45478716', 'GeneID:2767712']
```

Note：GenBank /EMBL 文件中的 gene 和 CDS features (join) 只包含外显子，不含内含子。

4.3.3 使用 feature 或 location 描述序列

`SeqFeature` 或 location object 对象并没有直接包含任何序列，只是可根据储存的 location (见第4.3.2节)，从父序列中取得。例如：某一短基因位于负链 5:18 位置，由于 GenBank/EMBL 文件以 1 开始计数，Biopython 中表示为 `complement(6..18)`：

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqFeature import SeqFeature, FeatureLocation
>>> example_parent = Seq("ACCGAGACGGCAAAGGCTAGCATAGGTATGAGACTTCCTTCCTGCCAGTGTGCTGAGGAAGTGGGAGCCTAC")
>>> example_feature = SeqFeature(FeatureLocation(5, 18), type="gene", strand=-1)
```

你可以用切片从父序列截取 5:18, 然后取反向互补序列。如果是 Biopython 1.59 或以后版本, 可使用如下方法:

```
>>> feature_seq = example_parent[example_feature.location.start:example_feature.location.end].reverse_complement()
>>> print feature_seq
AGCCTTTGCCGTC
```

不过在处理复合 features (joins) 时, 此法相当繁琐。此时可以使用 SeqFeature 对象的 extract 方法处理:

```
>>> feature_seq = example_feature.extract(example_parent)
>>> print feature_seq
AGCCTTTGCCGTC
```

SeqFeature 或 location 对象的长度等同于所表示序列的长度。

```
>>> print example_feature.extract(example_parent)
AGCCTTTGCCGTC
>>> print len(example_feature.extract(example_parent))
13
>>> print len(example_feature)
13
>>> print len(example_feature.location)
13
```

简单 FeatureLocation 对象的长度等于终止 position 减去起始 position 的差值; 而 CompoundLocation 的长度则为各片段长度之和。

4.4 References

对一条序列的注释还包括参考文献 (reference), Biopython 通过 Bio.SeqFeature.Reference 对象来储存相关的文献信息。

References 属性储存了 期刊名、题名、作者 等信息。此外还包括 medline_id、pubmed_id 以及 comment。

通常 reference 也有 location 对象, 便于文献涉及研究对象在序列中的定位。该 location 有可能是 一个 fuzzy location (见第4.3.2节)。

文献对象都以列表储存在 SeqRecord 对象的 annotations 字典中。字典的键为“references”。reference 对象也是为了方便处理文献而设计, 希望能满足各种使用需求。

4.5 格式化方法

SeqRecord 类中的 format() 能将字符串转换成被 Bio.SeqIO 支持的格式, 如 FASTA:

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein
```



```
record = SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD" \
    + "GAGAVIVGSDPDLSEVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPLISK" \
    + "NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRA TREVLSEYGNM" \
    + "SSAC", generic_protein),
    id="gi|14150838|gb|AAK54648.1|AF376133_1",
    description="chalcone synthase [Cucumis sativus]")

print record.format("fasta")
```

输出为:

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD
GAGAVIVGSDPDLSEVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRA TREVLSEYGNM
SSAC
```

`format` 方法接收单个必选参数, 小写字母字符串是 `Bio.SeqIO` 模块支持的输出格式 (见第5章)。然而, 此 `format()` 方法并不适用于包含多条序列的文件格式 (如多序列比对格式) (详见第5.5.4节)。

4.6 SeqRecord 切片

通过切片截取 `SeqRecord` 的部分序列可得到一条新的 `SeqRecord`。此处需引起注意的是 `per-letter annotations` 也被取切片, 但新序列中的 `features` 保持不变 (`locations` 相应调整)。

以前述 Genbank 文件为例:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")

>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])

>>> len(record)
9609
>>> len(record.features)
41
```

本例中, 我们关注 YP_pPCP05 质粒上的 `pim` 基因。从 GenBank 文件可直接看出 `pim` gene/CDS location 是 4343..4780 (相应的 Python 位置是 4342:4780)。Location 信息位于 GenBank 文件第 12 和 13 entries 中, 由于 python 以 0 开始计数, 因此 python 中, 它们是 `features` 列表中的 entries 11 和 12:

```
>>> print record.features[20]
type: gene
location: [4342:4780](+)
qualifiers:
  Key: db_xref, Value: ['GeneID:2767712']
  Key: gene, Value: ['pim']
  Key: locus_tag, Value: ['YP_pPCP05']

>>> print record.features[21]
type: CDS
location: [4342:4780](+)
```

qualifiers:

```
Key: codon_start, Value: ['1']
Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
Key: gene, Value: ['pim']
Key: locus_tag, Value: ['YP_pPCP05']
Key: note, Value: ['similar to many previously sequenced pesticin immunity ...']
Key: product, Value: ['pesticin immunity protein']
Key: protein_id, Value: ['NP_995571.1']
Key: transl_table, Value: ['11']
Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLNLTFGNSLSH...']
```

从父记录中取切片 (4300 到 4800), 观测所得到的 features 数量:

```
>>> sub_record = record[4300:4800]

>>> sub_record
SeqRecord(seq=Seq('ATAAATAGATTATTCCAAATAATTTATTTATGTAAGAACAGGATGGGAGGGGGA...TTA',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=[])

>>> len(sub_record)
500
>>> len(sub_record.features)
2
```

子记录 (sub_record) 只包括两个 features, 分别是 YP_pPCP05 质粒的 gene 和 CDS:

```
>>> print sub_record.features[0]
type: gene
location: [42:480](+)
qualifiers:
  Key: db_xref, Value: ['GeneID:2767712']
  Key: gene, Value: ['pim']
  Key: locus_tag, Value: ['YP_pPCP05']

>>> print sub_record.features[20]
type: CDS
location: [42:480](+)
qualifiers:
  Key: codon_start, Value: ['1']
  Key: db_xref, Value: ['GI:45478716', 'GeneID:2767712']
  Key: gene, Value: ['pim']
  Key: locus_tag, Value: ['YP_pPCP05']
  Key: note, Value: ['similar to many previously sequenced pesticin immunity ...']
  Key: product, Value: ['pesticin immunity protein']
  Key: protein_id, Value: ['NP_995571.1']
  Key: transl_table, Value: ['11']
  Key: translation, Value: ['MGGGMISKLFCLALIFLSSSGLAEKNTYTAKDILQNLNLTFGNSLSH...']
```

注意: locations 已被调整至对应生成的新父序列!

尽可能灵敏和直观地获取子记录的相关特征 (和任意的 per-letter annotation), 但是对于其余注释, Biopython 无法判断是否仍然适用于子记录。因此子记录忽略了 annotations 和 dbxrefs 以避免引起歧义。

```
>>> sub_record.annotations
{}
```

```
>>> sub_record.dbxrefs
[]
```

为了便于实际操作，子记录保留了 id, name 和 description：

```
>>> sub_record.id
'NC_005816.1'
>>> sub_record.name
'NC_005816'
>>> sub_record.description
'Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.'
```

上述例子很好的展示了问题，由于子记录不包括完整的质粒序列，因此 description 是错的。我们可以将子记录看做是截短版的 GenBank 文件，可用第4.5节中所述 format 方法纠正：

```
>>> sub_record.description = "Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, partial."
>>> print sub_record.format("genbank")
...
```

FASTQ 例子参见第18.1.7节和第18.1.8节（此例中 per-letter annotations (read 质量分数) 也被取切片）。

4.7 SeqRecord 对象相加

SeqRecord 对象可相加得到一个新的 SeqRecord。注意：per-letter annotations 也相加, features (locations 调整)；而其它 annotation 保持不变 (如 id、name 和 description)。

以 FASTQ 文件中的第一条记录为例说明 per-letter annotation（第5章详细介绍 SeqIO 函数）：

```
>>> from Bio import SeqIO
>>> record = SeqIO.parse("example.fastq", "fastq").next()
>>> len(record)
25
>>> print record.seq
CCCTTCTTGTCTTCAGCGTTCTCC

>>> print record.letter_annotations["phred_quality"]
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26,
26, 26, 26, 23, 23]
```

假设上述序列数据来自 Roche 454 测序，你根据其它信息得知 TTT 应该是 TT。此时可分别用切片提取第三个 T 前后的序列 (SeqRecord)：

```
>>> left = record[:20]
>>> print left.seq
CCCTTCTTGTCTTCAGCGTT
>>> print left.letter_annotations["phred_quality"]
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26]
>>> right = record[21:]
>>> print right.seq
CTCC
>>> print right.letter_annotations["phred_quality"]
[26, 26, 23, 23]
```

两部分相加：

```
>>> edited = left + right
>>> len(edited)
24
>>> print edited.seq
CCCTTCTTGTCTTCAGCGTTCCTCC

>>> print edited.letter_annotations["phred_quality"]
[26, 26, 18, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 22, 26, 26, 26, 26,
26, 26, 23, 23]
```

很容易和直观吧！上述两步可合并：

```
>>> edited = record[:20] + record[21:]
```

现在以 GenBank 文件（假定是环状基因组）为例说明 features：

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")

>>> record
SeqRecord(seq=Seq('TGTAACGAACGGTGCAATAGTGATCCACACCCAACGCCTGAAATCAGATCCAGG...CTG',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=['Project:10638'])

>>> len(record)
9609
>>> len(record.features)
41
>>> record.dbxrefs
['Project:58037']

>>> record.annotations.keys()
['comment', 'sequence_version', 'source', 'taxonomy', 'keywords', 'references',
'accessions', 'data_file_division', 'date', 'organism', 'gi']
```

可改变起点：

```
>>> shifted = record[2000:] + record[:2000]

>>> shifted
SeqRecord(seq=Seq('GATACGCAGTCATATTTTTTACACAATTCTCTAATCCCGACAAGGTCGTAGGTC...GGA',
IUPACAmbiguousDNA()), id='NC_005816.1', name='NC_005816',
description='Yersinia pestis biovar Microtus str. 91001 plasmid pPCP1, complete sequence.',
dbxrefs=[])

>>> len(shifted)
9609
```

Note: 上述方法并不完美（丢失了数据库交叉引用 dbxrefs 和源 feature）：

```
>>> len(shifted.features)
40
>>> shifted.dbxrefs
[]
>>> shifted.annotations.keys()
[]
```

这是因为 SeqRecord 切片对 annotation 保留非常谨慎（错误保留 annotation 可能引起大问题）。如果你想保留数据库的交叉引用 dbxrefs 和其余 annotations 字典必须明确说明，才能得以保留：

```
>>> shifted.dbxrefs = record.dbxrefs[:]
>>> shifted.annotations = record.annotations.copy()
>>> shifted.dbxrefs
['Project:10638']
>>> shifted.annotations.keys()
['comment', 'sequence_version', 'source', 'taxonomy', 'keywords', 'references',
'accessions', 'data_file_division', 'date', 'organism', 'gi']
```

Note: 此例中序列 record 的 identifiers 也应调整 (因为 NCBI 的 reference 链接的是未经修改的 原始序列)。

4.8 反向互补 SeqRecord 对象

为消除序列反向互补后 annotation 改变带来的困难, Biopython 1.57 SeqRecord 对象加入了 reverse_complement 方法。这也成为 Biopython 1.57 的新特性之一。

序列用 Seq 对象中的 reverse_complement 方法反向互补。Features 随 location 而改变, strand 也被重新计算。复制并反转 per-letter-annotation (通常情况下这种做法比较合适, 如对质量分数注释的反转)。然而多数 annotation 的转变却存有问题。

比如 record ID 是 accession 号, 该 accession 不应被用于反向互补序列。默认 identifier 转换可导致后续分析中的轻度数据损坏。因此 SeqRecord 的 id、name、description、annotations 和 dbxrefs 默认不变。

SeqRecord 对象的 reverse_complement 法用多个可选参数以对应 record 的属性。将这些参数设为 True 表示复制旧值; 而 False 意为用缺省值替换旧值。当然也可自定义新值。

举例:

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.gb", "genbank")
>>> print record.id, len(record), len(record.features), len(record.dbxrefs), len(record.annotations)
NC_005816.1 9609 41 1 11
```

反向互补该 record 并给 ID 赋予新值 - 注意: 多数 annotation 丢失, 而 features 仍在:

```
>>> rc = record.reverse_complement(id="TESTING")
>>> print rc.id, len(rc), len(rc.features), len(rc.dbxrefs), len(rc.annotations)
TESTING 9609 41 0 0
```


第 5 章序列输入和输出

本章将详细讨论 `Bio.SeqIO` 模块，该模块在第 2 章已经做过简单的介绍并在第 4 章使用过，它旨在提供一个简单的接口，实现对各种不同格式序列文件进行统一的处理。详细信息请查阅 `Bio.SeqIO` 维基页面 (<http://biopython.org/wiki/SeqIO>) 和内置文档 (`SeqIO`)：

```
>>> from Bio import SeqIO
>>> help(SeqIO)
...
```

学习本章的要领是学会使用 `SeqRecord` 对象（请见第 4 章 *<chapter-SeqRecord>* 章），该对象包含一个 `Seq` 对象（请见第 3 章）和注释信息（如序列 ID 和描述信息）。

5.1 解析/读取序列

该模块的主要函数是 `Bio.SeqIO.parse()`，它用于读取序列文件生成 `SeqRecord` 对象，包含两个参数：

1. 第一个参数是一个文件名或者一个句柄 (*handle*)。句柄可以是打开的文件，命令程序的输出，或者来自下载的数据（请见第 5.3 节）。更多关于句柄的信息请见第 22.1 节。
2. 第二个参数是一个小写字母字符串，用于指定序列格式（我们并不推测文件格式！），支持的文件格式请见 <http://biopython.org/wiki/SeqIO>。

还有一个用于指定字符集的 `alphabet` 参数，这对 FASTA 这样的文件格式非常有用，在这里 `Bio.SeqIO` 默认参数为字母表。

`Bio.SeqIO.parse()` 函数返回一个 `SeqRecord` 对象迭代器 (*iterator*)，迭代器通常用在循环中。

有时你需要处理只包含一个序列条目的文件，此时请使用函数 `Bio.SeqIO.read()`。它使用与函数 `Bio.SeqIO.parse()` 相同的参数，当文件有且仅有一个序列条目时返回一个 `SeqRecord` 对象，否则触发异常。

5.1.1 读取序列文件

总的来说，`Bio.SeqIO.parse()` 用于读取序列文件并返回 `SeqRecord` 对象，并通常用在循环中，如：

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
```

上面的示例来自第2.4节，它将读取来自 FASTA 格式文件 `ls_orchid.fasta` 的兰花 DNA 序列。如果你想读取 GenBank 格式文件，如 `ls_orchid.gbk`，只需要更改文件名和格式字符串：

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print seq_record.id
    print seq_record.seq
    print len(seq_record)
```

同样地，如果需要读取其他格式文件，并且 `Bio.SeqIO.parse()` 支持该文件格式，你只需要修改到相应的格式字符串，如“swiss”为 SwissProt 格式文件，“embl”为 EMBL 格式文本文件。详细的清单请见维基页面 (<http://biopython.org/wiki/SeqIO>) 和内置文档 ([在线文档](#))。

另外一个非常常见的使用 Python 迭代器的地方是在列表解析 (list comprehension，或者生成器表达式 generator expression)。例如，如果需要从文件中提取序列 ID 列表，我们可以通过以下的列表推导很容易地实现：

```
>>> from Bio import SeqIO
>>> identifiers = [seq_record.id for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank")]
>>> identifiers
['Z78533.1', 'Z78532.1', 'Z78531.1', 'Z78530.1', 'Z78529.1', 'Z78527.1', ..., 'Z78439.1']
```

更多关于 `SeqIO.parse()` 在列表推导中运用的示例请见第18.2节 (e.g. 对序列长度或 GC% 作图)。

5.1.2 遍历序列文件

在上述示例中，我们通常使用 `for` 循环遍历所有的序列条目 (records)。你可以对 `for` 循环使用所有类型的支持迭代接口的 Python 对象 (包括列表，元组 (tuple) 和字符串)。

`Bio.SeqIO` 返回的对象实际上是一个返回 `SeqRecord` 对象的迭代器。你将顺序地获得每个条目，但是有且仅有一次；优势是，当处理大文件时，迭代器可以有效地节约内存空间。

除了使用 `for` 循环，还可以使用迭代器的 `.next()` 方法遍历序列条目，如：

```
from Bio import SeqIO
record_iterator = SeqIO.parse("ls_orchid.fasta", "fasta")

first_record = record_iterator.next()
print first_record.id
print first_record.description

second_record = record_iterator.next()
print second_record.id
print second_record.description
```

注意：如果使用 `.next()` 方法，当没有序列条目时，将抛出 `StopIteration` 异常。

一种特殊情形是，序列文件包含多个序列条目，而你只需要第一个条目。在这种情况下，可使用以下代码，非常简洁：

```
from Bio import SeqIO
first_record = SeqIO.parse("ls_orchid.gbk", "genbank").next()
```

注意：像上述示例中使用 `.next()` 方法将忽略文件中其余的序列。如果序列文件“有且仅有”一条序列条目，如本章后面的某些在线示例、包含单条染色体序列的 GenBank 文件，请使用 `Bio.SeqIO.read()` 函数。该函数会检查文件是否包含额外的序列条目。

5.1.3 获得序列文件中序列条目列表

在上一节中，我们讨论了如何使用 `Bio.SeqIO.parse()` 返回一个 `SeqRecord` 迭代器，然后顺序地获取序列条目。往往我们需要以任意顺序获取序列条目，Python 列表数据类型便可以达到这个目的。使用 Python 内置函数 `list()`，我们可以将序列条目迭代器转变成 `SeqRecord` 对象列表，如下：

```
from Bio import SeqIO
records = list(SeqIO.parse("ls_orchid.gbk", "genbank"))

print "Found %i records" % len(records)

print "The last record"
last_record = records[-1] #using Python's list tricks
print last_record.id
print repr(last_record.seq)
print len(last_record)

print "The first record"
first_record = records[0] #remember, Python counts from zero
print first_record.id
print repr(first_record.seq)
print len(first_record)
```

运行结果:

```
Found 94 records
The last record
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT...GCC', IUPACAmbiguousDNA())
592
The first record
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
```

当然，你仍然可以对 `SeqRecord` 对象列表使用 `for` 循环。使用列表比使用迭代器灵活得多（例如，可以根据列表大小知道序列条目数量），但缺点是 `for` 循环要同时读取所有的内容，需要更多的内存空间。

5.1.4 提取数据

`SeqRecord` 对象及其注释信息在第4章中有更详细的介绍。为了解释注释信息是如何存储的，我们从 GenBank 文件 `ls_orchid.gbk` 中解析出第一个序列条目，并将其输出：

```
from Bio import SeqIO
record_iterator = SeqIO.parse("ls_orchid.gbk", "genbank")
first_record = record_iterator.next()
print first_record
```

输出结果:

```
ID: Z78533.1
Name: Z78533
Description: C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA.
Number of features: 5
/sequence_version=1
/source=Cypripedium irapeanum
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', ..., 'Cypripedium']
/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene', ..., 'ITS1', 'ITS2']
```

```
/references=[...]  
/accessions=['Z78533']  
/data_file.division=PLN  
/date=30-NOV-2006  
/organism=Cypripedium irapeanum  
/gi=2765658  
Seq('CGTAACAAGGTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
```

这可以得到 SeqRecord 大部分的易读的注释汇总信息。在此例中，我们将使用 .annotations 属性 -即 Python 字典 (dictionary)。该注释字典的内容如上述示例结果，你也可以直接输出：

```
print first_record.annotations
```

与其他 Python 字典一样，你可以轻松地获得键列表：

```
print first_record.annotations.keys()
```

或者值列表：

```
print first_record.annotations.values()
```

通常，注释值是字符串或者字符串列表。一个特例是，文件中的所有参考文献 (references) 都以引用 (reference) 对象方式存储。

例如你想从 GenBank 文件 `ls_orchid.gbk` 中提取出物种列表。我们需要的信息 *Cypripedium irapeanum* 被保存在这个注释字典的 'source' 和 'organism' 键中，我们可以用下面的方式获取：

```
>>> print first_record.annotations["source"]  
Cypripedium irapeanum
```

或：

```
>>> print first_record.annotations["organism"]  
Cypripedium irapeanum
```

通常，'organism' 用于学名 (拉丁名，e.g. *Arabidopsis thaliana*)，而 'source' 用于俗名 (common name) (e.g. thale cress)。在此例中，以及在通常情况下，这两个字段是相同的。

现在，让我们遍历所有的序列条目，创建一个包含所有兰花序列的物种列表：

```
from Bio import SeqIO  
all_species = []  
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):  
    all_species.append(seq_record.annotations["organism"])  
print all_species
```

另外一种方式是使用列表解析：

```
from Bio import SeqIO  
all_species = [seq_record.annotations["organism"] for seq_record in \  
                SeqIO.parse("ls_orchid.gbk", "genbank")]  
print all_species
```

两种方式的输出结果相同：

```
['Cypripedium irapeanum', 'Cypripedium californicum', ..., 'Paphiopedilum barbatum']
```

因为 GenBank 文件注释是以标准方式注释，所以相当简单。

现在，假设你需要从一个 FASTA 文件而不是 GenBank 文件提取出物种列表，那么你不得不多写一些代码，用以从序列条目的描述行提取需要的数据。使用的示例 FASTA 文件 `ls_orchid.fasta` 格式如下：

```
>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTTGATGAGACCGTGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
...
```

你可以手动检查，对于每一个序列条目，物种名都是描述行的第二个单词。这意味着如果我们以空白分割序列条目的 `.description`，物种名将会是第 1 个元素（第 0 个元素是序列 ID），我们可以这样做：

```
from Bio import SeqIO
all_species = []
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
    all_species.append(seq_record.description.split()[1])
print all_species
```

将得到：

```
['C.irapeanum', 'C.californicum', 'C.fasciculatum', 'C.margaritaceum', ..., 'P.barbatum']
```

使用更简洁的列表解析：

```
from Bio import SeqIO
all_species == [seq_record.description.split()[1] for seq_record in \
    SeqIO.parse("ls_orchid.fasta", "fasta")]
print all_species
```

通常，对 FASTA 描述行提取信息不是那么方便。如果你能获得对目标序列注释很好的文件格式如 GenBank 或者 EMBL，那么这类注释信息就很容易处理。

5.2 从压缩文档读取解析序列信息

在上一节中，我们研究了从文件中解析序列信息。除了使用文件名，你可以让 `Bio.SeqIO` 使用文件句柄（请见第 22.1 节）。在这一节，我们将使用文件句柄从压缩文件中解析序列信息。

正如你上面看到的，我们可以使用文件名作为 `Bio.SeqIO.read()` 或 `Bio.SeqIO.parse()` 的参数 - 例如在这个例子中，我们利用生成器表达式计算 GenBank 文件中多条序列条目的总长：

```
>>> from Bio import SeqIO
>>> print sum(len(r) for r in SeqIO.parse("ls_orchid.gb", "gb"))
67518
```

此处，我们使用文件句柄，并使用 `with` 语句（Python 2.5 及以上版本）自动关闭句柄：

```
>>> from __future__ import with_statement #Needed on Python 2.5
>>> from Bio import SeqIO
>>> with open("ls_orchid.gb") as handle:
...     print sum(len(r) for r in SeqIO.parse(handle, "gb"))
67518
```

或者，用旧版本的方式，手动关闭句柄：

```
>>> from Bio import SeqIO
>>> handle = open("ls_orchid.gb")
>>> print sum(len(r) for r in SeqIO.parse(handle, "gb"))
67518
>>> handle.close()
```

现在，如果我们有一个 `gzip` 压缩的文件呢？这种类型的文件在 Linux 系统中被普遍使用。我们可以使用 Python 的 `gzip` 模块打开压缩文档以读取数据 - 返回一个句柄对象：

```
>>> import gzip
>>> from Bio import SeqIO
>>> handle = gzip.open("ls_orchid.gbk.gz", "r")
>>> print sum(len(r) for r in SeqIO.parse(handle, "gb"))
67518
>>> handle.close()
```

相同地，如果我们有一个 bzip2 压缩文件（遗憾的是与函数的名字是不太一致）：

```
>>> import bz2
>>> from Bio import SeqIO
>>> handle = bz2.BZ2File("ls_orchid.gbk.bz2", "r")
>>> print sum(len(r) for r in SeqIO.parse(handle, "gb"))
67518
>>> handle.close()
```

如果你在使用 Python2.7 及以上版本，with 也可以读取 gzip 和 bz2 文件。然而在这之前的版本中使用将中断程序 (Issue 3860)，抛出 `_exit_` 缺失这类 属性错误 (AttributeError)。

有一种 gzip (GNU zip) 变种称为 BGZF (Blocked GNU Zip Format)，它可以作为普通 gzip 文件被读取，但具有随机读取的优点，我们将在稍后的第 5.4.4 节讨论。

5.3 解析来自网络的序列

在上一节中，我们研究了从文件（使用文件名或者文件句柄）和压缩文件（使用文件句柄）解析序列数据。这里我们将使用 Bio.SeqIO 的另一种类型句柄，网络连接，从网络下载和解析序列。

请注意，你可以一气呵成地下载序列并解析成为 SeqRecord 对象，这并不意味这是一个好主意。通常，你可能需要下载序列并存入文件以重复使用。

5.3.1 解析来自网络的 GenBank 序列条目

第 9.6 节将更详细地讨论 Entrez EFetch 接口，但是现在我们将通过它连接到 NCBI，通过 GI 号从 GenBank 获得 *Opuntia*（刺梨）序列。

首先，我们只获取一条序列条目。如果你不关注注释和相关信息，下载 FASTA 文件是个不错的选择，因为他们相对紧凑。请记住，当你希望处理的对象包含有且仅有一条序列条目时，使用 Bio.SeqIO.read() 函数：

```
from Bio import Entrez
from Bio import SeqIO
Entrez.email = "A.N.Other@example.com"
handle = Entrez.efetch(db="nucleotide", rettype="fasta", retmode="text", id="6273291")
seq_record = SeqIO.read(handle, "fasta")
handle.close()
print "%s with %i features" % (seq_record.id, len(seq_record.features))
```

输出结果为:

```
gi|6273291|gb|AF191665.1|AF191665 with 0 features
```

NCBI 也允许你获取其它格式文件，尤其是 GenBank 文件。直到 2009 年复活节，Entrez EFetch API 使用“genbank”作为返回类型。然而 NCBI 现在坚持使用“gb”（蛋白使用“gp”）作为官方返回类型，具体描述参见 [EFetch for Sequence and other Molecular Biology Databases](#)。因此，Biopython1.50 及以后版本的 Bio.SeqIO 中，我们支持“gb”作为“genbank”的别名。

```

from Bio import Entrez
from Bio import SeqIO
Entrez.email = "A.N.Other@example.com"
handle = Entrez.efetch(db="nucleotide", rettype="gb", retmode="text", id="6273291")
seq_record = SeqIO.read(handle, "gb") #using "gb" as an alias for "genbank"
handle.close()
print "%s with %i features" % (seq_record.id, len(seq_record.features))

```

输出结果为：

```
AF191665.1 with 3 features
```

请注意，这次我们获得 3 个特征。

现在，让我们获取多个序列条目。这次句柄包含多条序列条目，因此我们必须使用 `Bio.SeqIO.parse()` 函数：

```

from Bio import Entrez
from Bio import SeqIO
Entrez.email = "A.N.Other@example.com"
handle = Entrez.efetch(db="nucleotide", rettype="gb", retmode="text", \
                       id="6273291,6273290,6273289")
for seq_record in SeqIO.parse(handle, "gb"):
    print seq_record.id, seq_record.description[:50] + "..."
    print "Sequence length %i," % len(seq_record),
    print "%i features," % len(seq_record.features),
    print "from: %s" % seq_record.annotations["source"]
handle.close()

```

输出结果为：

```

AF191665.1 Opuntia marenae rpl16 gene; chloroplast gene for c...
Sequence length 902, 3 features, from: chloroplast Opuntia marenae
AF191664.1 Opuntia clavata rpl16 gene; chloroplast gene for c...
Sequence length 899, 3 features, from: chloroplast Grusonia clavata
AF191663.1 Opuntia bradtiana rpl16 gene; chloroplast gene for...
Sequence length 899, 3 features, from: chloroplast Opuntia bradtiana

```

更多关于 `Bio.Entrez` 模块的信息请见第 9 章，并阅读 NCBI Entrez 使用指南（第 9.1 节）。

5.3.2 解析来自网络的 SwissProt 序列条目

现在我们使用句柄下载来自 ExPASy 的 SwissProt 文件，更深入的信息请见第 10 章。如上面提到的，当你希望处理的对象包含有且仅有一条序列条目时，使用 `Bio.SeqIO.read()` 函数：

```

from Bio import ExPASy
from Bio import SeqIO
handle = ExPASy.get_sprot_raw("023729")
seq_record = SeqIO.read(handle, "swiss")
handle.close()
print seq_record.id
print seq_record.name
print seq_record.description
print repr(seq_record.seq)
print "Length %i" % len(seq_record)
print seq_record.annotations["keywords"]

```

如果网络连接正常，你将会得到：

```
023729
CHS3_BROFI
RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone synthase 3;
Seq('MAPAMEEIRQAQRAEGPAAVLAIGTSTPPNALYQADYPDYFRITKSEHLTELK...GAE', ProteinAlphabet())
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']
```

5.4 序列文件作为字典

我们将介绍 Bio.SeqIO 模块中 3 个相关函数，用于随机读取多序列文件。这里需要权衡灵活性和内存使用。总之：

- Bio.SeqIO.to_dict() 最灵活但内存占用最大（请见第 5.4.1 节）。这基本上是一个辅助函数，用于建立 Python 字典，每个条目以 SeqRecord 对象形式存储在内存中，允许你修改这些条目。
- Bio.SeqIO.index() 处于中间水平，类似于只读字典，当需要时解析序列到 SeqRecord 对象（请见第 5.4.2 节）。
- Bio.SeqIO.index_db() 也类似于只读字典，但是将文件中的 ID 和文件偏移值存储到硬盘（SQLite3 数据库），这意味着它对内存需求很低（请见第 5.4.3 节），但会慢一点。

全面的概述请见讨论部分（第 5.4.5 节）。

5.4.1 序列文件作为字典 - 在内存中

我们对兰花数据文件接下来的处理将用于展示如何对他们建立索引，以及使用 Python 的 dictionary 数量类型（与 Perl 中 hash 类似）以类似于数据库的方式读取数据。这常用于从中等大小的文件中读取某些特定元素，形成一个很好的快速数据库。如果处理较大的文件，内存将是个问题，请见下面第 5.4.2 节。

你可以使用 Bio.SeqIO.to_dict() 函数创建一个 SeqRecord 字典（在内存中）。默认会使用每条序列条目的 ID（i.e. .id 属性）作为键。让我们用 GenBank 文件试一试：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.gbk", "genbank"))
```

Bio.SeqIO.to_dict() 仅需一个参数，即能够得到 SeqRecord 对象的列表或生成器，这里我们使用 SeqIO.parse 函数输出。顾名思义，Bio.SeqIO.to_dict() 返回一个 Python 字典。

因为变量 orchid_dict 是一个普通的 Python 字典，我们可以查看所有的键：

```
>>> len(orchid_dict)
94

>>> print orchid_dict.keys()
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

如果你确实需要，你甚至可以一次性查看所有的序列条目：

```
>>> orchid_dict.values() #lots of output!
...
```

我们可以通过键读取单个 SeqRecord 对象并操作改对象：

```
>>> seq_record = orchid_dict["Z78475.1"]
>>> print seq_record.description
P.supardii 5.8S rRNA gene and ITS1 and ITS2 DNA.
```

```
>>> print repr(seq_record.seq)
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT', IUPACAmbiguousDNA())
```

因此，可以用我们的 GenBank 序列条目轻松地在内存中创建一个数据库 (in memory “database”)。接下来我们将尝试使用 FASTA 文件。

值得注意的是，对有 Python 使用经验的人来说，可以轻松地创建一个类似的字典。然而，典型的字典构建方法不能很好地处理重复键的情况。使用 `Bio.SeqIO.to_dict()` 函数将明确检查重复键，如果发现任何重复键将引发异常并退出。

5.4.1.1 指定字典键

使用上述相同的代码，仅将文件改为 FASTA 文件：

```
from Bio import SeqIO
orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.fasta", "fasta"))
print orchid_dict.keys()
```

这次键为：

```
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
..., 'gi|2765613|emb|Z78488.1|PTZ78488', 'gi|2765583|emb|Z78458.1|PHZ78458']
```

这结果是之前在第 2.4.1 节中我们解析的 FASTA 文件结果。如果你需要别的作为键，如登录号 (Accession Number)，可使用 `SeqIO.to_dict()` 的可选参数 `key_function`，它允许你根据你的序列条目特点，自定义字典键。

首先，你必须写一个函数，当使用 `SeqRecord` 对象作为参数时，可以返回你需要的键 (字符串)。通常，函数的细节依赖于你要处理的序列条目的特点。但是对于我们的兰花数据，我们只需要使用“管道”符号 (|) 切分 ID 并返回第四个条目 (第三个元素)：

```
def get_accession(record):
    """Given a SeqRecord, return the accession number as a string.

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = record.id.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]
```

然后我们可以将此函数赋与 `SeqIO.to_dict()` 函数用于构建字典：

```
from Bio import SeqIO
orchid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.fasta", "fasta"), key_function=get_accession)
print orchid_dict.keys()
```

最终可到到新的字典键：

```
>>> print orchid_dict.keys()
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

不是太困难！

5.4.1.2 使用 SEGUID 校验和对字典建立索引

为了介绍另外一个 `SeqRecord` 对象字典的示例，我们将使用 SEGUID 校验和函数。这是一个相对较新的校验和，冲突非常罕见 (i.e. 两条不同序列具有相同的校验和)，相对 CRC64 校验和有所提升。

让我们再一次处理兰花 GenBank 文件：

```
from Bio import SeqIO
from Bio.SeqUtils.CheckSum import seguid
for record in SeqIO.parse("ls_orchid.gb", "genbank"):
    print record.id, seguid(record.seq)
```

将得到：

```
Z78533.1 JUEoWn6DPHgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439.1 H+JfaShya/4yyAj7IbMqgNkxdxQ
```

现在，再次调用 `Bio.SeqIO.to_dict()` 函数 `key_function` 参数，`key_function` 参数需要一个函数将 `SeqRecord` 转变为字符串。我们不能直接使用 `seguid()` 函数，因为它需要 `Seq` 对象（或字符串）作为参数。不过，我们可以使用 Python 的 `lambda` 特性创建一个一次性（“one off”）函数，然后传递给 `Bio.SeqIO.to_dict()`：

```
>>> from Bio import SeqIO
>>> from Bio.SeqUtils.CheckSum import seguid
>>> seguid_dict = SeqIO.to_dict(SeqIO.parse("ls_orchid.gb", "genbank"),
...                               lambda rec : seguid(rec.seq))
>>> record = seguid_dict["MN/s0q9zDoCVEEc+k/IFwCNF2pY"]
>>> print record.id
Z78532.1
>>> print record.description
C.californicum 5.8S rRNA gene and ITS1 and ITS2 DNA.
```

将会返回文件中第二个序列条目 `Z78532.1`。

5.4.2 序列文件作为字典 - 索引文件

之前众多示例试图解释的是使用 `Bio.SeqIO.to_dict()` 的灵活性。然而，因为它将所有的信息都存储在内存中，你能处理的文件大小受限于电脑的 RAM。通常，这仅能处理一些小文件或中等大小文件。

对于更大的文件，应该考虑使用 `Bio.SeqIO.index()`，工作原理上略有不同。尽管仍然是返回一个类似于字典的对象，它并不将所有的信息存储在内存中。相反，它仅仅记录每条序列条目在文件中的位置 - 当你需要读取某条特定序列条目时，它才进行解析。

让我们使用之前相同的 GenBank 文件作为示例：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gb", "genbank")
>>> len(orchid_dict)
94

>>> orchid_dict.keys()
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']

>>> seq_record = orchid_dict["Z78475.1"]
>>> print seq_record.description
P.supardii 5.8S rRNA gene and ITS1 and ITS2 DNA.
>>> seq_record.seq
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGATCACAT...GGT', IUPACAmbiguousDNA())
```

注意：`Bio.SeqIO.index()` 不接受句柄参数，仅仅接受文件名。这有充分的理由，但是过于技术性。第二个参数是文件格式（与其它 `Bio.SeqIO` 函数一样的小写字串）。你可以使用许多其他的简单的文件

格式，包括 FASTA 和 FASTQ 文件（示例参见第 18.1.11 节），但不支持比对文件格式，如 PHYLIP 或 Clustal。最后有个可选参数，你可以指定字符集或者键函数。

下面是使用 FASTA 文件做的相同的示例 - 仅改变了文件名和格式：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.fasta", "fasta")
>>> len(orchid_dict)
94
>>> orchid_dict.keys()
['gi|2765596|emb|Z78471.1|PDZ78471', 'gi|2765646|emb|Z78521.1|CCZ78521', ...
..., 'gi|2765613|emb|Z78488.1|PTZ78488', 'gi|2765583|emb|Z78458.1|PHZ78458']
```

5.4.2.1 指定字典键

如果想使用与之前一样的键，像第 5.4.1.1 节 Bio.SeqIO.to_dict() 示例，你需要写一个小函数，从 FASTA ID（字符串）中匹配你想要的键：

```
def get_acc(identifier):
    """Given a SeqRecord identifier string, return the accession number as a string.

    e.g. "gi|2765613|emb|Z78488.1|PTZ78488" -> "Z78488.1"
    """
    parts = identifier.split("|")
    assert len(parts) == 5 and parts[0] == "gi" and parts[2] == "emb"
    return parts[3]
```

然后我们将此函数赋与 Bio.SeqIO.index() 函数用于构建字典：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.fasta", "fasta", key_function=get_acc)
>>> print orchid_dict.keys()
['Z78484.1', 'Z78464.1', 'Z78455.1', 'Z78442.1', 'Z78532.1', 'Z78453.1', ..., 'Z78471.1']
```

当你知道怎样实现就变得很简单了。

5.4.2.2 获取序列条目原始数据

来自 Bio.SeqIO.index() 的字典样对象以 SeqRecord 对象形式返回序列条目。但是，有时候从文件中直接获取原始数据非常有用。对于此种情况，使用 get_raw() 方法，它仅需要一个参数（序列 ID），然后返回一个字符串（提取自文件的未处理数据）。

一个重要的例子就是从大文件中提取出一个序列子集，特别是当 Bio.SeqIO.write() 还不支持这种输出格式（e.g. SwissProt 文件格式的文本文件）或者需要完整地保留源文件（Biopython 的 GenBank 和 EMBL 格式输出并不会保留每一点注释信息）。

假如你已经从 UniProt FTP 站点下载了整个数据库的 SwissPort 格式文本文件 (ftp://ftp.uniprot.org/pub/databases/uniprot/current-release/knowledgebase/complete/uniprot_sprot.dat.gz)，并也已经解压为文件 uniprot_sprot.dat，你需要从中提取一部分序列条目：

```
>>> from Bio import SeqIO
>>> uniprot = SeqIO.index("uniprot_sprot.dat", "swiss")
>>> handle = open("selected.dat", "w")
>>> for acc in ["P33487", "P19801", "P13689", "Q8JZQ5", "Q9TRC7"]:
...     handle.write(uniprot.get_raw(acc))
>>> handle.close()
```

在第18.1.5节有更多关于使用 `SeqIO.index()` 函数对大文件序列排序的示例（不需要一次加载所有信息到内存）。

5.4.3 序列文件作为字典 - 数据库索引文件

Biopython 1.57 引入一个替代的函数，`Bio.SeqIO.index_db()`。由于它将序列信息以文件方式存储在硬盘上（使用 SQLite3 数据库）而不是内存中，因此它可以处理超大文件。同时，你可以同时对多个文件建立索引（前提是所有序列条目的 ID 是唯一的）。

`Bio.SeqIO.index()` 函数有三个参数：

- 索引文件名，我们建议使用以 `.idx` 结尾的字符，改索引文件实质上是 SQLite3 数据库；
- 要建立索引的文件列表（或者单个文件名）；
- 文件格式（与 `SeqIO` 模块中其它函数一样的小写字符串）。

将以 NCBI FTP 站点 <ftp://ftp.ncbi.nih.gov/genbank/> 的 GenBank 文本文件为例，这些文件为 gzip 压缩文件。对于 GenBank 版本 182，病毒序列共包含 16 个文件，`gbvrl1.seq` - `gbvrl16.seq`，共包含约一百万条序列条目。对这些文件，你可以像这样建立索引：

```
>>> from Bio import SeqIO
>>> files = ["gbvrl%i.seq" % (i+1) for i in range(16)]
>>> gb_vrl = SeqIO.index_db("gbvrl.idx", files, "genbank")
>>> print "%i sequences indexed" % len(gb_vrl)
958086 sequences indexed
```

在我个人电脑上，运行大约需要 2 分钟。如果你重新运行，索引文件（这里为 `gbvrl.idx`）将在不到一秒的时间内加载。你可以将这个索引作为一个只读的 Python 字典，并不需要去担心序列来自哪个文件，e.g.:

```
>>> print gb_vrl["GQ333173.1"].description
HIV-1 isolate F12279A1 from Uganda gag protein (gag) gene, partial cds.
```

5.4.3.1 获取序列条目原始数据

与第5.4.2.2节讨论的 `Bio.SeqIO.index()` 函数一样，该字典样对象同样允许你获取每个序列条目的原始文件：

```
>>> print gb_vrl.get_raw("GQ333173.1")
LOCUS      GQ333173              459 bp    DNA        linear    VRL 21-OCT-2009
DEFINITION HIV-1 isolate F12279A1 from Uganda gag protein (gag) gene, partial
            cds.
ACCESSION   GQ333173
...
//
```

5.4.4 对压缩文件建立索引

经常你要建立索引的文件可能非常大，因此你想对它进行压缩。不幸的是，对常规的文件格式如 `gzip` 和 `bzip2` 高效的随机读取通常很困难。在这种情况下，BGZF (Blocked GNU Zip Format) 非常有用。它是 `gzip` 变体（也可以使用标准的 `gzip` 工具解压），因 BAM 文件格式得到推广，`samtools` 和 `tabix`；

你可以使用 `samtools` 的命令行工具 `bgzip` 创建 BGZF 格式压缩文件。在我们的示例中，使用文件扩展名 `*.bgz`，以区别于普通的压缩文件（命名为 `*.gz`）。你也可以在 Python 中使用 `Bio.bgzf` 模块读写 BGZF 文件。

`Bio.SeqIO.index()` 和 `Bio.SeqIO.index_db()` 函数均可以用于 BGZF 压缩文件。例如，如果使用过未压缩的 GenBank 文件：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk", "genbank")
>>> len(orchid_dict)
94
```

你可以使用如下的命令行命令压缩该文件（同时保留源文件） - 不需要担心，压缩文件和别的示例及已经包含：

```
$ bgzip -c ls_orchid.gbk > ls_orchid.gbk.bgz
```

你可以用相同的方式使用压缩文件：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index("ls_orchid.gbk.bgz", "genbank")
>>> len(orchid_dict)
94
```

或：

```
>>> from Bio import SeqIO
>>> orchid_dict = SeqIO.index_db("ls_orchid.gbk.bgz.idx", "ls_orchid.gbk.bgz", "genbank")
>>> len(orchid_dict)
94
```

`SeqIO` 建立索引时自动检测是否为 BGZF 压缩格式。注意：压缩文件和未压缩文件不能使用相同的索引文件。

5.4.5 讨论

这些方法你该使用哪种及其原因，取决于你要做什么（以及你要处理的数据有多大）。然而，通常 `Bio.SeqIO.index()` 是个不错的选择。如果你正在处理上百万条序列条目，多个文件，或者重复性分析，那么看看 `Bio.SeqIO.index_db()`。

选择 `Bio.SeqIO.to_dict()` 而不选择 `Bio.SeqIO.index()` 或 `Bio.SeqIO.index_db()` 的原因主要是它的灵活性，尽管会占用更多内存。存储 `SeqRecord` 对象到内存的优势在于可以随意被改变，添加或者删除。除了高内存消耗这个缺点外，建立索引也可能花费更长的时间，因为所有的条目都需要被完全解析。

`Bio.SeqIO.index()` 和 `Bio.SeqIO.index_db()` 都是在需要时才解析序列条目。当建立索引时，他们扫描文件，寻找每个序列条目的起始，并做尽可能少的工作提取出 ID 信息。

选择 `Bio.SeqIO.index()` 而不选择 `Bio.SeqIO.index_db()` 的原因包括以下：

- 建立索引更快（需要注意的是简单文件格式）
- 读取 `SeqRecord` 对象稍快（但是这种差异只有在解析简单格式文件是才可见）
- 可以使用不可变的 Python 对象作为字典键而不仅仅是字符串（e.g. 如字符串元组、不可变容器（frozen set））
- 如果被建立索引的序列文件改变，不需要担心索引数据库过期。

选择 `Bio.SeqIO.index_db()` 而不选择 `Bio.SeqIO.index()` 的原因包括以下：

- 没有内存限制 - 这对通常多达 10 亿的二代测序文件来说非常重要，如果使用 `Bio.SeqIO.index()` 可能需要超过 4G 的 RAM 和 64 位 Python
- 索引数据量保存在硬盘上，可重复使用。尽管建立索引数据库需要花费更多的时间，但是从长远看来。如果你有个脚本重新运行这个相同的数据库，可以节约时间

- 可以同时多个文件建立索引
- `get_raw()` 方法可以快得多，因为对于大多数文件格式只需要存储序列条目的长度和偏移量 (offset)

5.5 写入序列文件

我们已经讨论了使用 `Bio.SeqIO.parse()` 输入序列 (读取文件)，现在我们将研究使用 `Bio.SeqIO.write()` 输出序列 (写入文件)。该函数需要三个参数：某些 `SeqRecord` 对象，要写入的句柄或文件名，和序列格式。

我们先使用硬编码方式 (手动创建而不是从文件中加载) 创建一些新的 `SeqRecord` 对象，示例如下：

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein

rec1 = SeqRecord(Seq("MMYQQGCFAGGTVLRRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD" \
    + "GAGAVIVGSDPDLSEVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPLISK" \
    + "NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRA TREVLSEYGNM" \
    + "SSAC", generic_protein),
    id="gi|14150838|gb|AAK54648.1|AF376133_1",
    description="chalcone synthase [Cucumis sativus]")

rec2 = SeqRecord(Seq("YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYL TEEILKENPSMCEYMAPSLDARQ" \
    + "DMVVVEIPKLGKEAAVKAIKEWGQ", generic_protein),
    id="gi|13919613|gb|AAK33142.1|",
    description="chalcone synthase [Fragaria vesca subsp. bracteata]")

rec3 = SeqRecord(Seq("MVTVEEFRRQAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC" \
    + "EKSMIKKRYMHL TEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQKAIKEWGQP" \
    + "KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLRMAKDLAEN" \
    + "NKGARVLVVCSEITAVTFRGPNDTHLDSL VGQALFGDGAAAVIIGSDPIPEVERPLFELV" \
    + "SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPLISK NIEKSLVEAFQPLGISDWNLSFW" \
    + "IAHPGGPAILDQVELKLGKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT" \
    + "TGEGLWGVLF GFGPGLTVETVVLHSVAT", generic_protein),
    id="gi|13925890|gb|AAK49457.1|",
    description="chalcone synthase [Nicotiana tabacum]")

my_records = [rec1, rec2, rec3]
```

现在我们得到一个 `SeqRecord` 对象列表，将它写入一个 FASTA 格式文件：

```
from Bio import SeqIO
SeqIO.write(my_records, "my_example.faa", "fasta")
```

如果你喜欢的文本编辑软件打开，可得到：

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSMVGQALFGD
GAGAVIVGSDPDLSEVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKDVPLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRA TREVLSEYGNM
SSAC
>gi|13919613|gb|AAK33142.1| chalcone synthase [Fragaria vesca subsp. bracteata]
YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYL TEEILKENPSMCEYMAPSLDARQ
DMVVVEIPKLGKEAAVKAIKEWGQ
>gi|13925890|gb|AAK49457.1| chalcone synthase [Nicotiana tabacum]
MVTVEEFRRQAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELKEKFKRMC
```

```
EKSMIKKRYMHLTEELKENPNICAYMAPSLDARQDIVVEVPKLGKEAAQKAIKEWGQP
KSKITHLVFCTTSGVDMPCDYQLTKLLGLRPSVKRFMYQQGCFAGGTVLRMAKDLAEN
NKGARVLVVCSEITAVTRGPNDTLDSL VGQALFGDGA AAVIIGSDPIPEVERPLFELV
SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPLISKNIEKSLVEAFQPLGISDWNLSFW
IAHPGGPAILDQVELKGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKASAKEGLGT
TGEGLWGVLF GFGPGLTVETVVLHSVAT
```

怎样才能知道 `Bio.SeqIO.write()` 函数写入了多少条序列条目到句柄呢？如果你的序列条目保存在一个列表中，只需要使用 `len(my_records)`，但是你不能对来自生成器/迭代器的序列条目。`Bio.SeqIO.write()` 函数本身就返回写入文件的 `SeqRecord` 对象个数。

- 注意 - 如果你 `Bio.SeqIO.write()` 函数要写入的文件已经存在，旧文件将会被覆写，并且不会得到任何警告信息。

5.5.1 可逆读写 (Round trips)

某些人需要他们的解析器是“可逆”的，即当你读入某个文件后可以按原样写回。这需要解析器提取足够多的信息用于 * 精确 * 还原原始文件，`Bio.SeqIO` 不打算这么做。

一个简单的例子是，FASTA 文件中，允许序列以任意字符数换行。解析以下两条序列得到一个相同的 `SeqRecord` 对象，这两条序列仅在换行上不同：

```
>YAL068C-7235.2170 Putative promoter sequence
TACGAGAATAATTTCTCATCATCCAGCTTTAACACAAAATTCGCACAGTTTTCGTTAAGA
GAACCTTAACATTTCTTATGACGTAAATGAAGTTTATATATAAATTCCTTTTATTGGA
```

```
>YAL068C-7235.2170 Putative promoter sequence
TACGAGAATAATTTCTCATCATCCAGCTTTAACACAAAATTCGCA
CAGTTTTCGTTAAGAGAACTTAACATTTTCTTATGACGTAAATGA
AGTTTATATATAAATTCCTTTTATTGGA
```

为了创建一个可逆读写的 FASTA 解析器，需要记录序列换行发生的位置，而这些额外的信息通常毫无意义。因此，`Biopython` 在输出时使用默认的 60 字符换行。空白字符在许多其他文件格式中运用也存在相同的问题。另外一个问题是，在某些情况下，`Biopython` 并不能保存每一点注释信息 (e.g. `GenBank` 和 `EMBL`)。

少数时候，重要的是保留原来的布局（这可能有点怪异），第 5.4.2.2 节关于 `Bio.SeqIO.index()` 字典样对象的 `get_raw()` 方法提供了可能的解决方案。

5.5.2 序列格式间的转换

在之前的例子中我们使用 `SeqRecord` 对象列表作为 `Bio.SeqIO.write()` 函数的输入，但是它也接受来自于 `Bio.SeqIO.parse()` 的 `SeqRecord` 迭代器 - 这允许我们通过结合使用这两个函数实现文件转换。

在这个例子中，我们将读取 `GenBank` 格式文件 `ls_orchid.gbk`，然后输出为 FASTA 格式文件：

```
from Bio import SeqIO
records = SeqIO.parse("ls_orchid.gbk", "genbank")
count = SeqIO.write(records, "my_example.fasta", "fasta")
print "Converted %i records" % count
```

这仍然有点复杂，因为文件格式转换是比较常见的任务，有一个辅助函数可以替代上述代码：

```
from Bio import SeqIO
count = SeqIO.convert("ls_orchid.gbk", "genbank", "my_example.fasta", "fasta")
print "Converted %i records" % count
```

Bio.SeqIO.convert() 函数可以使用句柄或文件名。然而需要注意的是，如果输出文件已存在，将覆盖该文件。想了解更多信息，请使用内置帮助文档：

```
>>> from Bio import SeqIO
>>> help(SeqIO.convert)
...
```

原理上讲，只需要改变文件名和格式字符串，该代码即可实现 Biopython 支持的文件格式间的转换。然而，写入某种格式时需要某些特定的信息 (e.g. 质量值)，而其他格式文件不包含此信息。例如，你可以将 FASTQ 转化为 FASTA 文件，却不能进行逆操作。不同 FASTQ 格式间的相互转变请见 cookbook 章第 18.1.9 节和第 18.1.10 节。

最后，使用 Bio.SeqIO.convert() 函数额外的好处是更快，(最大的好处是代码会更短) 原因是该转换函数可以利用几个文件格式特殊的优化条件和技巧。

5.5.3 转化序列到反向互补序列

假设你有一个核苷酸序列文件，需要转换成一个包含其反向互补的文件。这时，需要做些工作，将从文件得到的 SeqRecord 对象转化为适合存储到输出文件的信息。

首先，我们将使用 Bio.SeqIO.parse() 加载文件中的核酸序列，然后使用 Seq 对象的内置方法 .reverse_complement() 输出其反向互补序列 (请见第 3.7 节)。

```
>>> from Bio import SeqIO
>>> for record in SeqIO.parse("ls_orchid.gb", "genbank"):
...     print record.id
...     print record.seq.reverse_complement()
```

现在，如果我们想保存这些反向互补序列到某个文件，需要创建 SeqRecord 对象。我们可以使用 SeqRecord 对象的内置方法 .reverse_complement() (请见第 4.8 节)，但是我们必须决定新的序列条目怎么命名。

这是一个绝好的展示列表解析效率地方，列表解析通过在内存中创建一个列表实现：

```
>>> from Bio import SeqIO
>>> records = [rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
...             for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
>>> len(records)
```

这时就用到了列表解析的绝妙之处，在其中添加一个条件语句：

```
>>> records = [rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
...             for rec in SeqIO.parse("ls_orchid.fasta", "fasta") if len(rec)<700]
>>> len(records)
18
```

这将在内存中创建一个序列小于 700bp 的反向互补序列列表。我们可以以相同的方式使用生成器表达式 - 但是更有优势的是，它不需要同时在内存中创建所有序列条目的列表：

```
>>> records = (rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
...             for rec in SeqIO.parse("ls_orchid.fasta", "fasta") if len(rec)<700)
```

完整的示例如下：

```
>>> from Bio import SeqIO
>>> records = (rec.reverse_complement(id="rc_"+rec.id, description = "reverse complement") \
...             for rec in SeqIO.parse("ls_orchid.fasta", "fasta") if len(rec)<700)
>>> SeqIO.write(records, "rev_comp.fasta", "fasta")
18
```


在第18.1.3节有一个相关的示例，将 FASTA 文件中核酸序列翻译为氨基酸。

5.5.4 获得格式化为字符串的 SeqRecord 对象

有时你不需要将序列条目写入文件或者句柄，而是想获得包含特定格式序列条目的字符串。Bio.SeqIO 接口基于句柄，但是 Python 有一个有用的内置模块，提供基于字符串的句柄。

举个例子来说明如果使用这个功能，我们先从兰花 GenBank 文件加载一系列 SeqRecord 对象，然后创建一个包含 FASTA 格式序列条目的字符串：

```
from Bio import SeqIO
from StringIO import StringIO
records = SeqIO.parse("ls_orchid.gbk", "genbank")
out_handle = StringIO()
SeqIO.write(records, out_handle, "fasta")
fasta_data = out_handle.getvalue()
print fasta_data
```

当你第一次看到，会觉得这并不够简单明了。在特殊情况下，你希望得到一个只包含特定格式的单条序列条目的字符串，可以使用 SeqRecord 类的 format()（请见第4.5节）。

注意：尽管我们不鼓励这么做，你可以使用 format() 方法写入文件，示例如下：

```
from Bio import SeqIO
out_handle = open("ls_orchid_long.tab", "w")
for record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    if len(record) > 100:
        out_handle.write(record.format("tab"))
out_handle.close()
```

这类代码可以处理顺序文件格式如 FASTA 或者此处使用的简单的制表符分割文件，但不能处理更复杂的或是交错式文件格式。这就是为什么我们仍然强调使用 Bio.SeqIO.write() 的原因，如下面的示例：

```
from Bio import SeqIO
records = (rec for rec in SeqIO.parse("ls_orchid.gbk", "genbank") if len(rec) > 100)
SeqIO.write(records, "ls_orchid.tab", "tab")
```

同时，单次调用 SeqIO.write(...) 也比多次调用 SeqRecord.format(...) 方法更快。

第 6 章多序列比对

多序列比对 (Multiple Sequence Alignment, MSA) 是指对多个序列进行对位排列。这通常需要保证序列间的等同位点处在同一列上, 并通过引进小横线 (-) 以保证最终的序列具有相同的长度。这种序列比对可以视作是由字符组成的矩阵。在 Biopython 中, 多序列比对中每一个序列是以 SeqRecord 对象来表示的。

这里我们介绍一种新的对象 - MultipleSeqAlignment 来表示这样一类数据, 我们还将介绍 Bio.AlignIO 模块来读写不同格式的多序列比对数据 (Bio.AlignIO 在设计上与之前介绍的 Bio.SeqIO 模块是类似的)。Biopython 中, Bio.SeqIO 和 Bio.AlignIO 都能读写各种格式的多序列比对数据。在实际处理中, 使用哪一个模块取决于用户需要对数据进行何种操作。

本章的最后一部分是关于各种常用多序列比对程序 (ClustalW 和 MUSCLE) 的 Biopython 命令行封装。

6.1 读取多序列比对数据

在 Biopython 中, 有两种方法读取多序列比对数据, Bio.AlignIO.read() 和 Bio.AlignIO.parse()。这两种方法跟 Bio.SeqIO 处理一个和多个数据的设计方式是一样的。Bio.AlignIO.read() 只能读取一个多序列比对而 Bio.AlignIO.parse() 可以依次读取多个序列比对数据。

使用 Bio.AlignIO.parse() 将会返回一个 MultipleSeqAlignment 的迭代器 (iterator)。迭代器往往在循环中使用。在实际数据分析过程中会时常处理包含有多个多序列比对的文件。例如 PHYLIP 中的 seqboot, EMBOSS 工具箱中的 water 和 needle, 以及 Bill Pearson 的 FASTA 程序。

然而在大多数情况下, 你所遇到的文件仅仅包括一个多序列比对。这时, 你应该使用 Bio.AlignIO.read(), 这将返回一个 MultipleSeqAlignment 对象。

这两个函数都接受两个必须参数:

1. 第一个参数为包含有多序列比对数据的句柄 (handle)。在实际操作中, 这往往是一个打开的文件 (详细信息请见 22.1) 或者文件名。
2. 第二个参数为多序列比对文件格式 (小写)。与 Bio.SeqIO 模块一样, Biopython 不会对将读取的文件格式进行猜测。所有 Bio.AlignIO 模块支持的多序列比对数据格式可以在 <http://biopython.org/wiki/AlignIO> 中找到。

Bio.AlignIO 模块还接受一个可选参数 seq_count。这一参数将在 6.1.3 中具体讨论。它可以处理不确定的多序列比对格式, 或者包含有多个序列的排列。

另一个可选参数 alphabet 允许用户指定序列比对文件的字符 (alphabet), 它可以用来说明序列比对的类型 (DNA, RNA 或蛋白质)。因为大多数序列比对格式并不区别序列的类型, 因此指定这一参数可能会对后期的分析产生帮助。Bio.AlignIO 默认将使用一般字符 (generic alphabet), 这将不区分各种序列比对类型。

6.1.1 单一的序列比对

例如，请见以下 PFAM（或者 Stockholm）格式的蛋白序列比对文件。

```
# STOCKHOLM 1.0
#=GS COATB_BPIKE/30-81 AC P03620.1
#=GS COATB_BPIKE/30-81 DR PDB; 1if1 ; 1-52;
#=GS Q9TOQ8_BPIKE/1-52 AC Q9TOQ8.1
#=GS COATB_BPI22/32-83 AC P15416.1
#=GS COATB_BPM13/24-72 AC P69541.1
#=GS COATB_BPM13/24-72 DR PDB; 2cpb ; 1-49;
#=GS COATB_BPM13/24-72 DR PDB; 2cps ; 1-49;
#=GS COATB_BPZJ2/1-49 AC P03618.1
#=GS Q9TOQ9_BPF1/1-49 AC Q9TOQ9.1
#=GS Q9TOQ9_BPF1/1-49 DR PDB; 1nh4 A; 1-49;
#=GS COATB_BPIF1/22-73 AC P03619.2
#=GS COATB_BPIF1/22-73 DR PDB; 1ifk ; 1-50;
COATB_BPIKE/30-81 AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLVIRLFKKFSSKA
#=GR COATB_BPIKE/30-81 SS -HHHHHHHHHHHHHH--HHHHHHHH--HHHHHHHHHHHHHHHHHHHHHHHHHHHH---
Q9TOQ8_BPIKE/1-52 AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLVIRLFKKFVSRA
COATB_BPI22/32-83 DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTTSVAVAGLAIRLFKKFSSKA
COATB_BPM13/24-72 AEGDDP...AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR COATB_BPM13/24-72 SS ---S-T...CHCHHHHCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHHHCTT--
COATB_BPZJ2/1-49 AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
Q9TOQ9_BPF1/1-49 AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
#=GR Q9TOQ9_BPF1/1-49 SS -----...HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
COATB_BPIF1/22-73 FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
#=GR COATB_BPIF1/22-73 SS XX-HHHH--HHHHHH--HHHHHH--HHHHHHHHHHHHHHHHHHHHHHHHHHHH---
#=GC SS_cons XHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHC--
#=GC seq_cons AEssss...AptAhDSLpspAT-hIu.sWshVsslVsAsluIKLFKKFsSKA
//
```

这是 PFAM 数据库中 Phage_Coat_Gp8 的种子排列 (PF05371)。该排列下载于一个已经过期的 PFAM 数据库版本 (<http://pfam.sanger.ac.uk/>)，但这并不影响我们的例子（假设你已经将以上内容下载到一个名为“PF05371_seed.sth”的文件中，并在 Python 的当前工作目录下）：

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
```

这段代码将在屏幕上打印出该序列比对的概要信息：

```
>>> print alignment
SingleLetterAlphabet() alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVAGLVIRL...SRA Q9TOQ8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9TOQ9_BPF1/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIK...SRA COATB_BPIF1/22-73
```

你会注意到，以上输出截短了中间一部分序列的内容。你也可以很容易地通过控制多序列比对中每一条序列（作为 SeqRecord 对象）来输出你所喜欢的格式。例如：

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print "Alignment length %i" % alignment.get_alignment_length()
Alignment length 52
>>> for record in alignment:
```

```
...     print "%s - %s" % (record.seq, record.id)
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAGLVIRLFKKFVSRA - Q9TOQ8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTPVVTTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9TOQ9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

你也可以使用上面 alignment 对象的 format 方法来以指定的格式显示它。具体信息可以参见 6.2.2。

你是否已经注意到以上原始数据文件中包含有蛋白数据库 (PDB) 交叉引用以及相关二级结构的信息？你可以尝试以下代码：

```
>>> for record in alignment:
...     if record.dbxrefs:
...         print record.id, record.dbxrefs
COATB_BPIKE/30-81 ['PDB; 1ifl ; 1-52;']
COATB_BPM13/24-72 ['PDB; 2cpb ; 1-49;', 'PDB; 2cps ; 1-49;']
Q9TOQ9_BPFD/1-49 ['PDB; 1nh4 A; 1-49;']
COATB_BPIF1/22-73 ['PDB; 1ifk ; 1-50;']
```

如果你希望显示所有的序列注释信息，请使用以下例子：

```
>>> for record in alignment:
...     print record
```

Sanger 网站 <http://pfam.sanger.ac.uk/family?acc=PF05371> 可以让你下载各种不同的序列比对的格式。以下例子为 FASTA 格式：

```
>COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAGLVIRLFKKFSSKA
>Q9TOQ8_BPIKE/1-52
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAGLVIRLFKKFVSRA
>COATB_BPI22/32-83
DGTSTATSYATEAMNSLKTQATDLIDQTPVVTTSVAVAGLAIRLFKKFSSKA
>COATB_BPM13/24-72
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
>Q9TOQ9_BPFD/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA
```

注意 Sanger 网站有一个选项可以将序列比对中的间隔 (gap) 用小圆点或者是小横线表示。在以上例子中，序列间隔由小横线表示。假设你已经下载该文件，并保存为“PF05371_seed.faa”。你可以使用以下代码来读入该序列比对。

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.faa", "fasta")
print alignment
```

你可能已经发现，以上代码中唯一的变化只是指定格式的参数。所返回的 alignment 对象将会包含同样的序列和序列名字。但是仔细的读者会发现，每一个 alignment 的 SeqRecord 中并不包含数据的引用注释。这是因为 FASTA 格式本身并没有包含这一类信息。

此外，除了使用 Sanger 网站，你也可以利用 Bio.AlignIO 来将原始的 Stockholm 格式转换成 FASTA 文件格式（见下文）。

对于任何一种 Biopython 支持的格式，你都可以用同样的方式读取它（通过指定文件的格式）。例如，你可以使用“`phylip`”来表示 PHYLIP 格式文件，用“`nexus`”来指定 NEXUS 格式文件或者用“`emboss`”来指定 EMBOSS 工具箱的输出文件。读者可以在以下链接中找到所有支持的格式 (<http://biopython.org/wiki/AlignIO>)，或者内置的帮助中（以及在线文档 [online](#)）：

```
>>> from Bio import AlignIO
>>> help(AlignIO)
...
```

6.1.2 多个序列比对

在前一章中，我们旨在读取仅包含有一个序列比对的文件。然而，在很多情况下，文件可能包含有多个序列比对。这时，你可以使用 `Bio.AlignIO.parse()` 来读取它们。

假设我们有一个 PHYLIP 格式的很小的序列比对：

```
5      6
Alpha   AACCAAC
Beta    AACCCC
Gamma   ACCCAAC
Delta   CCACCA
Epsilon CCAAAC
```

如果你想用 PHYLIP 工具包来 bootstrap 一个系统发生树，其中的一个步骤是用 `bootseq` 程序来产生许多序列比对。这将给出类似于以下格式的序列比对：

```
5      6
Alpha   AAACCA
Beta    AAACCC
Gamma   ACCCCA
Delta   CCCAAC
Epsilon CCCAAA

5      6
Alpha   AAACAA
Beta    AAACCC
Gamma   ACCCAA
Delta   CCCACC
Epsilon CCCAAA

5      6
Alpha   AAAAAC
Beta    AAACCC
Gamma   AACCAAC
Delta   CCCCCA
Epsilon CCCAAC
...

5      6
Alpha   AAAACC
Beta    ACCCCC
Gamma   AAAACC
Delta   CCCCAA
Epsilon CAAACC
```

如果你想用 `Bio.AlignIO` 来读取这个文件，你可以使用：

```
from Bio import AlignIO
alignments = AlignIO.parse("resampled.phy", "phylip")
for alignment in alignments:
```

```
print alignment
print
```

这将给出以下的输出（这时只显示缩略的一部分）：

```
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACCA Alpha
AAACCC Beta
ACCCCA Gamma
CCCAAC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAACAA Alpha
AAACCC Beta
ACCCAA Gamma
CCCAAC Delta
CCCAAA Epsilon

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACCAAC Gamma
CCCCCA Delta
CCCAAC Epsilon

...

SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

与 `Bio.SeqIO.parse` 一样，`Bio.SeqIO.parse()` 将返回一个迭代器（iterator）。如果你希望把所有的序列比对都读取到内存中，以下代码将把它们储存在一个列表对象里。

```
from Bio import AlignIO
alignments = list(AlignIO.parse("resampled.phy", "phylip"))
last_align = alignments[-1]
first_align = alignments[0]
```

6.1.3 含糊的序列比对

许多序列比对的文件格式可以非常明确地储存多个序列比对。然而，例如 FASTA 一类的普通序列文件格式并没有很直接的分隔符来分开多个序列比对。读者可以见以下例子：

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Beta
```

```
ACTACCGCTAGCTCAGAAG
>Gamma
ACTACGGCTAGCACAGAAG
```

以上 FASTA 格式文件可以认为是一个包含有 6 条序列的序列比对（有重复序列名）。或者从文件名来看，这很可能是两个序列比对，每一个包含有三个序列，只是这两个序列比对恰好具有相同的长度。

以下是另一个例子：

```
>Alpha
ACTACGACTAGCTCAG--G
>Beta
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Gamma
ACTACGGCTAGCACAGAAG
>Alpha
ACTACGACTAGCTCAGG--
>Delta
ACTACGGCTAGCACAGAAG
```

同样，这也可能是一个包含有六个序列的序列比对。然而，根据序列名判断，这很可能是三个两两间的序列比较，而且恰好有同样的长度。

最后一个例子也类似：

```
>Alpha
ACTACGACTAGCTCAG--G
>XXX
ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG
```

在这一个例子中，由于序列有不同的长度，这不能被当作是一个包含六个序列的单独的序列比对。很显然，这可以被看成是三个两两间的序列比对。

很明显，将多个序列比对以 FASTA 格式储存并不方便。然而，在某些情况下，如果你一定要这么做，Bio.AlignIO 依然能够处理上述情形（但是所有的序列比对必须都含有相同的序列）。一个很常见的例子是，我们经常会使用 EMBOSS 工具箱中的 needle 和 water 来产生许多两两间的序列比对——然而在这种情况下，你可以指定数据格式为“emboss”，Bio.AlignIO 仍然能够识别这些原始输出。

为了处理这样的 FASTA 格式的数据，我们可以指定 Bio.AlignIO.parse() 的第三个可选参数 seq_count，这一参数将告诉 Biopython 你所期望的每个序列比对中序列的个数。例如：

```
for alignment in AlignIO.parse(handle, "fasta", seq_count=2):
    print "Alignment length %i" % alignment.get_alignment_length()
    for record in alignment:
        print "%s - %s" % (record.seq, record.id)
    print
```

这将给出：

```

Alignment length 19
ACTACGACTAGCTCAG--G - Alpha
ACTACCGCTAGCTCAGAAG - XXX

Alignment length 17
ACTACGACTAGCTCAGG - Alpha
ACTACGGCAAGCACAGG - YYY

Alignment length 21
--ACTACGAC--TAGCTCAGG - Alpha
GGACTACGACAATAGCTCAGG - ZZZ

```

如果你使用 `Bio.AlignIO.read()` 或者 `Bio.AlignIO.parse()` 而不指定 `seq_count`，这将返回一个包含有六条序列的序列比对。对于上面的第三个例子，由于序列长度不同，导致它们不能被解析为一个序列比对，Biopython 将会抛出一个异常。

如果数据格式本身包含有分割符，`Bio.AlignIO` 可以很聪明地自动确定文件中每一个序列比对，而无需指定 `seq_count` 选项。如果你仍然指定 `seq_count` 但是却与数据本身的分隔符相冲突，Biopython 将产生一个错误。

注意指定这一可选的 `seq_count` 参数将假设文件中所有的序列比对都包含相同数目的序列。假如你真的遇到每一个序列比对都有不同数目的序列，`Bio.AlignIO` 将无法读取。这时，我们建议你使用 `Bio.SeqIO` 来读取数据，然后将序列转换为序列比对。

6.2 序列比对的写出

我们已经讨论了 `Bio.AlignIO.read()` 和 `Bio.AlignIO.parse()` 来读取各种格式的序列比对，现在让我们来使用 `Bio.AlignIO.write()` 写出序列比对文件。

这一函数接受三个参数：一个 `MultipleSeqAlignment` 对象（或者是一个 `Alignment` 对象），一个可写的文件句柄（handle）或者期望写出的文件名，以及写出文件的格式。

这里有一个手动构造一个 `MultipleSeqAlignment` 对象的例子（注意 `MultipleSeqAlignment` 是由若干个 `SeqRecord` 组成的）：

```

from Bio.Alphabet import generic_dna
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Align import MultipleSeqAlignment

align1 = MultipleSeqAlignment([
    SeqRecord(Seq("ACTGCTAGCTAG", generic_dna), id="Alpha"),
    SeqRecord(Seq("ACT-CTAGCTAG", generic_dna), id="Beta"),
    SeqRecord(Seq("ACTGCTAGDTAG", generic_dna), id="Gamma"),
])

align2 = MultipleSeqAlignment([
    SeqRecord(Seq("GTCAGC-AG", generic_dna), id="Delta"),
    SeqRecord(Seq("GACAGCTAG", generic_dna), id="Epsilon"),
    SeqRecord(Seq("GTCAGCTAG", generic_dna), id="Zeta"),
])

align3 = MultipleSeqAlignment([
    SeqRecord(Seq("ACTAGTACAGCTG", generic_dna), id="Eta"),
    SeqRecord(Seq("ACTAGTACAGCT-", generic_dna), id="Theta"),
    SeqRecord(Seq("-CTACTACAGGTG", generic_dna), id="Iota"),
])

```



```
my_alignments = [align1, align2, align3]
```

现在我们有一个包含三个 `MultipleSeqAlignment` 对象的列表 (`my_alignments`)，现在我们将它写为 PHYLIP 格式：

```
from Bio import AlignIO
AlignIO.write(my_alignments, "my_example.phy", "phylip")
```

如果你用你喜欢的文本编辑器在你当前的工作目录下打开 `my_example.phy` 文件，你会看到以下内容：

```
3 12
Alpha      ACTGCTAGCT AG
Beta       ACT-CTAGCT AG
Gamma      ACTGCTAGDT AG
3 9
Delta      GTCAGC-AG
Epsilon    GACAGCTAG
Zeta       GTCAGCTAG
3 13
Eta        ACTAGTACAG CTG
Theta      ACTAGTACAG CT-
Iota       -CTACTACAG GTG
```

在更多情况下，你希望读取一个已经含有序列比对的文件，经过某些操作（例如去掉一些行和列）然后将它重新储存起来。

假如你希望知道有多少序列比对被 `Bio.AlignIO.write()` 函数写入句柄中。如果你的序列比对都被放在一个列表中（如同以上的例子），你可以很容易地使用 `len(my_alignments)` 来获得这一信息。然而，如果你的序列比对在一个生成器/迭代器对象中，你无法轻松地完成这件事情。为此，`Bio.AlignIO.write()` 将会返回它所写出的序列比对个数。

注意 - 如果你所指定给 `Bio.AlignIO.write()` 的文件已经存在在当前目录下，这一文件将被直接覆盖掉而不会有任何警告。

6.2.1 序列比对的格式间转换

`Bio.AlignIO` 模块中的序列比对格式转换功能与 `Bio.SeqIO`（见 5.5.2）模块的格式转换是一样的。在通常情况下，我们建议使用 `Bio.AlignIO.parse()` 来读取序列比对数据，然后使用 `Bio.AlignIO.write()` 函数来写出。或者你也可以直接使用 `Bio.AlignIO.convert()` 函数来实现格式的转换。

在本例中，我们将读取 PFAM/Stockholm 格式的序列比对，然后将其保存为 Clustal 格式：

```
from Bio import AlignIO
count = AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.aln", "clustal")
print "Converted %i alignments" % count
```

或者，使用 `Bio.AlignIO.parse()` 和 `Bio.AlignIO.write()`：

```
from Bio import AlignIO
alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")
count = AlignIO.write(alignments, "PF05371_seed.aln", "clustal")
print "Converted %i alignments" % count
```

`Bio.AlignIO.write()` 函数默认处理的情形是一个包括有多个序列比对的对象。在以上例子中，我们给予 `Bio.AlignIO.write()` 的参数是一个由 `Bio.AlignIO.parse()` 函数返回的一个迭代器。

在以下例子中，我们知道序列比对文件中仅包含有一个序列比对，因此我们使用 `Bio.AlignIO.read()` 函数来读取数据，然后使用 `Bio.AlignIO.write()` 来将数据保存为另一种格式：

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
AlignIO.write([alignment], "PF05371_seed.aln", "clustal")
```

使用以上两个例子，你都可以将 PFAM/Stockholm 格式的序列比对数据转换为 Clustal 格式：

CLUSTAL X (1.81) multiple sequence alignment

```
COATB.BPIKE/30-81      AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSS
Q9TOQ8.BPIKE/1-52     AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVS
COATB.BPI22/32-83     DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTTSVAVAGLAIRLFKKFSS
COATB.BPM13/24-72     AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFVS
COATB.BPZJ2/1-49      AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFAS
Q9TOQ9.BPFD/1-49      AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFVS
COATB.BPIF1/22-73     FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVS

COATB.BPIKE/30-81      KA
Q9TOQ8.BPIKE/1-52     RA
COATB.BPI22/32-83     KA
COATB.BPM13/24-72     KA
COATB.BPZJ2/1-49      KA
Q9TOQ9.BPFD/1-49      KA
COATB.BPIF1/22-73     RA
```

另外，你也可以使用以下代码将它保存为 PHYLIP 格式：

```
from Bio import AlignIO
AlignIO.convert("PF05371_seed.sth", "stockholm", "PF05371_seed.phy", "phylip")
```

你可以获得以下 PHYLIP 格式的文件输出：

```
7 52
COATB.BPIK AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTVVVVAGL VIRLFKKFSS
Q9TOQ8.BPI AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTVVVVAGL VIKLFKKFVS
COATB.BPI2 DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB.BPM1 AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFVS
COATB.BPZJ AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFAS
Q9TOQ9.BPF AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVIVGATI GIKLFKKFVS
COATB.BPIF FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

KA
RA
KA
KA
KA
KA
KA
RA
```

PHYLIP 格式最大的一个缺陷就是它严格地要求每一条序列的 ID 是都为 10 个字符（ID 中多出的字符将被截短）。在这一个例子中，截短的序列 ID 依然是唯一的（只是缺少了可读性）。在某些情况下，我们并没有一个好的方式去压缩序列的 ID。以下例子提供了另一种解决方案——利用自定义的序列 ID 来代替原本的序列 ID：

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
name_mapping = {}
```

```
for i, record in enumerate(alignment):
    name_mapping[i] = record.id
    record.id = "seq%i" % i
print name_mapping

AlignIO.write([alignment], "PF05371_seed.phy", "phylip")
```

以上代码将会建立一个字典对象实现自定义的 ID 和原始 ID 的映射：

```
{0: 'COATB_BPIKE/30-81', 1: 'Q9T0Q8_BPIKE/1-52', 2: 'COATB_BPI22/32-83', ...}
```

以下为 PHYLIP 的格式输出：

```
7 52
seq0      AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
seq1      AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
seq2      DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
seq3      AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFVS
seq4      AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
seq5      AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFVS
seq6      FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS

      KA
      RA
      KA
      KA
      KA
      KA
      RA
```

由于序列 ID 的限制性，PHYLIP 格式不是储存序列比对的理想格式。我们建议你将数据储存成 PFAM/Stockholm 或者其它能对序列比对进行注释的格式来保存你的数据。

6.2.2 将序列比对对象转换为格式化字符串 (formatted strings)

因为 Bio.AlignIO 模块是基于文件句柄的，因此你如果想将序列比对读入为一个字符串对象，你需要做一些额外的工作。然而，我们提供一个 format() 方法来帮助你实现这项任务。format() 方法需要用户提供一个小写的格式参数（这可以是任何 AlignIO 支持的序列比对格式）。例如：

```
from Bio import AlignIO
alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
print alignment.format("clustal")
```

我们在 4.5 中讲到，Bio.SeqIO 也有一个对 SeqRecord 输出的方法。

format() 方法是利用 StringIO 以及 Bio.AlignIO.write() 来实现以上输出的。如果你使用的是较老版本的 Biopython，你可以使用以下代码来完成相同的工作：

```
from Bio import AlignIO
from StringIO import StringIO

alignments = AlignIO.parse("PF05371_seed.sth", "stockholm")

out_handle = StringIO()
AlignIO.write(alignments, out_handle, "clustal")
clustal_data = out_handle.getvalue()

print clustal_data
```

6.3 序列比对的操纵

现在我们已经了解了如何读入和写出序列比对。让我们继续看看如何对读入的序列比对进行操作。

6.3.1 序列比对的切片 (slice) 操作

首先，用户可以认为读入的序列比对是一个由 SeqRecord 对象构成的 Python 列表 (list)。有了这样一个印象以后，你可以使用 len() 方法来得到行数 (序列比对的个数)，你也可以对序列比对进行迭代。

```
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> print "Number of rows: %i" % len(alignment)
Number of rows: 7
>>> for record in alignment:
...     print "%s - %s" % (record.seq, record.id)
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAVGLVIRLFKKFSSKA - COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAVGLVIKLFKKFVSRA - Q9TOQ8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTPVVTTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9TOQ9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73
```

你可以使用列表所拥有的 append 和 extend 方法来给序列比对增加序列。请读者一定要正确理解序列比对与其包含的序列的关系，这样你就可以使用切片操作来获得其中某些序列比对。

```
>>> print alignment
SingleLetterAlphabet() alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAVGLVIRL...SKA COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTPVVTTVVAVGLVIKL...SRA Q9TOQ8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTPVVTTSVAVAGLAIRL...SKA COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9TOQ9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
>>> print alignment[3:7]
SingleLetterAlphabet() alignment with 4 rows and 52 columns
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA Q9TOQ9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA COATB_BPIF1/22-73
```

假如你需要获得特定的列该怎么办呢？如果你接触过 Numpy 矩阵那么一定对下面的语法非常熟悉，使用双切片：

```
>>> print alignment[2,6]
T
```

使用两个整数来获得序列比对中的一个字符，这其实是以下操作的简化方式：

```
>>> print alignment[2].seq[6]
T
```

你可以用下面的代码来获取整列：

```
>>> print alignment[:,6]
TTT---T
```

你也可以同时选择特定的行和列。例如，以下代码将打印出第 3 到 6 行的前 6 列：

```
>>> print alignment[3:6,:6]
SingleLetterAlphabet() alignment with 3 rows and 6 columns
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9TOQ9_BPF1/1-49
```

使用：将打印出所有行：

```
>>> print alignment[:,6]
SingleLetterAlphabet() alignment with 7 rows and 6 columns
AEPNAA COATB_BPIKE/30-81
AEPNAA Q9TOQ8_BPIKE/1-52
DGTSTA COATB_BPI22/32-83
AEGDDP COATB_BPM13/24-72
AEGDDP COATB_BPZJ2/1-49
AEGDDP Q9TOQ9_BPF1/1-49
FAADDA COATB_BPIF1/22-73
```

切片给我们提供了一个简单的方式来去除一部分序列比对。在以下例子中，有三条序列的 7, 8, 9 三列为间隔 (-)。

```
>>> print alignment[:,6:9]
SingleLetterAlphabet() alignment with 7 rows and 3 columns
TNY COATB_BPIKE/30-81
TNY Q9TOQ8_BPIKE/1-52
TSY COATB_BPI22/32-83
--- COATB_BPM13/24-72
--- COATB_BPZJ2/1-49
--- Q9TOQ9_BPF1/1-49
TSQ COATB_BPIF1/22-73
```

你也可以通过切片来获得第 9 列以后的所有序列：

```
>>> print alignment[:,9:]
SingleLetterAlphabet() alignment with 7 rows and 43 columns
ATEAMDSLKTQAIDLSQTWPVVTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
ATEAMDSLKTQAIDLSQTWPVVTVVVAGLVIRLFKKFVSRA Q9TOQ8_BPIKE/1-52
ATEAMNSLKTQATDLIDQTWPVVTSAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AKAAFNLSLQASATEYIGYAWAMVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AKAAFDLSLQASATEYIGYAWAMVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AKAAFDLSLQASATEYIGYAWAMVVIVGATIGIKLFKKFTSKA Q9TOQ9_BPF1/1-49
AKAAFDLSLQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

现在，你可以通过列来操纵序列比对。这也是你能够去除序列比对中的许多列。例如：

```
>>> edited = alignment[:,6] + alignment[:,9:]
>>> print edited
SingleLetterAlphabet() alignment with 7 rows and 49 columns
AEPNAAATEAMDSLKTQAIDLSQTWPVVTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEPNAAATEAMDSLKTQAIDLSQTWPVVTVVVAGLVIRLFKKFVSRA Q9TOQ8_BPIKE/1-52
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
AEGDDPAKAANSLQASATEYIGYAWAMVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAANFDLSLQASATEYIGYAWAMVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEGDDPAKAANFDLSLQASATEYIGYAWAMVVIVGATIGIKLFKKFTSKA Q9TOQ9_BPF1/1-49
FAADDAKAANFDLSLQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
```

另一个经常使用的序列比对操作是将多个基因的序列比对拼接成一个大的序列比对 (meta-alignment)。在进行这种操作时一定要注意序列的 ID 需要匹配 (具体请见 4.7 关于 SeqRecord 的说明)。为了达到这种

目的，用 `sort()` 方法将序列 ID 按照字母顺序进行排列可能会有所帮助：

```
>>> edited.sort()
>>> print edited
SingleLetterAlphabet() alignment with 7 rows and 49 columns
DGTSTAATEAMNSLKTQATDLIDQTWPVVTSTVAVAGLAIRLFKKFSSKA COATB_BPI22/32-83
FAADDAAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA COATB_BPIF1/22-73
AEPNAAATEAMDSLKTQAIDLISQTPVVTVVVAGLVIRLFKKFSSKA COATB_BPIKE/30-81
AEGDDPAKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA COATB_BPM13/24-72
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA COATB_BPZJ2/1-49
AEPNAAATEAMDSLKTQAIDLISQTPVVTVVVAGLVIKLFKKFVSRA Q9TOQ8_BPIKE/1-52
AEGDDPAKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA Q9TOQ9_BPFD/1-49
```

注意：只有当两个序列比对拥有相同的行的时候才能进行序列比对的拼接。

6.3.2 序列比对作为数组

根据你的需要，有时将序列比对转换为字符数组是非常方便的。你可以用 Numpy 来实现这一目的：

```
>>> import numpy as np
>>> from Bio import AlignIO
>>> alignment = AlignIO.read("PF05371_seed.sth", "stockholm")
>>> align_array = np.array([list(rec) for rec in alignment], np.character)
>>> align_array.shape
(7, 52)
```

如果你需要频繁地使用列操作，你可以让 Numpy 将序列比对以列的形式进行储存（与 Fortran 一样），而不是 Numpy 默认形式（与 C 一样以行储存）：

```
>>> align_array = np.array([list(rec) for rec in alignment], np.character, order="F")
```

注意，Numpy 的数组和 Biopython 默认的序列比对对象是分别储存在内存中的，编辑其中的一个不会更新另一个的值。

6.4 构建序列比对的工具

目前有非常多的算法来帮助你构建一个序列比对，包括两两间的比对和多序列比对。这些算法在计算上往往是非常慢的，你一定不会希望用 Python 来实现他们。然而，你可以使用 Biopython 来运行命令行程序。通常你需要：

1. 准备一个包含未比对序列的输入文件，一般为 FASTA 格式的序列。你可以使用 `Bio.SeqIO` 来创建一个（具体见第 5 章）。
2. 在 Biopython 中运行一个命令行程序来构建序列比对（我们将在这里详细介绍）。这需要通过 Biopython 的打包程序（wrapper）来实现。
3. 读取以上程序的输出，也就是排列好的序列比对。这往往可以通过 `Bio.AlignIO` 来实现（请看本章前部分内容）。

本章所介绍的所有的命令行打包程序都将以同样的方式使用。你创建一个命令行对象来指定各种参数（例如：输入文件名，输出文件名等），然后通过 Python 的系统命令模块来运行这一程序（例如：使用 `subprocess` 进程）。

大多数的打包程序都在 `Bio.Align.Applications` 中定义：

```
>>> import Bio.Align.Applications
>>> dir(Bio.Align.Applications)
...
['ClustalwCommandline', 'DialignCommandline', 'MafftCommandline', 'MuscleCommandline',
'PrankCommandline', 'ProbconsCommandline', 'TCoffeeCommandline' ...]
```

(以下划线开头的记录不是 Biopython 打包程序，这些变量在 Python 中有特殊的含义。) Bio.Emboss.Applications 中包含对 EMBOSS 的打包程序 (包括 needle 和 water)。EMBOSS 和 PHYLIP 的打包程序将在 6.4.5 节中详细介绍。在本章中，我们并不打算将所有的序列比对程序都予以介绍，但是 Biopython 中各种序列比对程序都具有相同的使用方式。

6.4.1 ClustalW

ClustalW 是一个非常流行的进行多序列比对的命令程序 (其还有一个图形化的版本称之为 ClustalX)。Biopython 的 Bio.Align.Applications 模块包含这一多序列比对程序的打包程序。

我们建议你在 Python 中使用 ClustalW 之前在命令行界面下手动使用 ClustalW，这样能使你更清楚这一程序的参数。你会发现 Biopython 打包程序非常严格地遵循实际的命令行 API：

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> help(ClustalwCommandline)
...
```

作为最简单的一个例子，你仅仅需要一个 FASTA 格式的序列文件作为输入，例如：opuntia.fasta (你可以在线或者在 Biopython/Doc/examples 文件夹中找到该序列)。opuntia.fasta 包含着 7 个 prickly-pear 的 DNA 序列 (来自仙人掌科)。

ClustalW 在默认情况下会产生一个包括所有输入序列的序列比对以及一个由输入序列名字构成的指导树 (guide tree)。例如，用上述文件作为输入，ClustalW 将会输出 opuntia.aln 和 opuntia.dnd 两个文件：

```
>>> from Bio.Align.Applications import ClustalwCommandline
>>> cline = ClustalwCommandline("clustalw2", infile="opuntia.fasta")
>>> print cline
clustalw2 -infile=opuntia.fasta
```

注意这里我们给出的执行文件名是 clustalw2，这是 ClustalW 的第二个版本 (第一个版本的文件名为 clustalw)。ClustalW 的这两个版本具有相同的参数，并且在功能上也是一致的。

你可能会发现，尽管你安装了 ClustalW，以上的命令行却无法正确运行。你可能会得到“command not found”的错误信息 (尤其是在 Windows 上)。这往往是由于 ClustalW 的运行程序并不在系统的工作目录 PATH 下 (一个包含着运行程序路径的环境变量)。你既可以修改 PATH，使其包括 ClustalW 的运行程序 (不同系统需要以不同的方式修改)，或者你也可以直接指定程序的绝对路径。例如：

```
>>> import os
>>> from Bio.Align.Applications import ClustalwCommandline
>>> clustalw_exe = r"C:\Program Files\new clustal\clustalw2.exe"
>>> clustalw_cline = ClustalwCommandline(clustalw_exe, infile="opuntia.fasta")

>>> assert os.path.isfile(clustalw_exe), "Clustal W executable missing"
>>> stdout, stderr = clustalw_cline()
```

注意，Python 中 \n 和 \t 会被解析为一个新行和制表空白 (tab)。然而，如果你将一个小写的“r”放在字符串的前面，这一字符串就将保留原始状态，而不被解析。这种方式对于指定 Windows 风格的文件名来说是一种良好的习惯。

Biopython 在内部使用较新的 subprocess 模块来实现打包程序，而不是 os.system() 和 os.popen*。

现在，我们有必要去了解命令行工具是如何工作的。当你使用一个命令行时，它往往会在屏幕上输出一些内容。这一输出可以被保存或重定向。在系统输出中，有两种管道（pipe）来区分不同的输出信息—标准输出（standard output）包含正常的输出内容，标准错误（standard error）显示错误和调试信息。同时，系统也接受标准输入（standard input）。这也是命令行工具如何读取数据文件的。当程序运行结束以后，它往往会返回一个整数。一般返回值为 0 意味着程序正常结束。

当你使用 Biopython 打包程序来调用命令行工具的时候，它将会等待程序结束，并检查程序的返回值。如果返回值不为 0，Biopython 将会提示一个错误信息。Biopython 打包程序将会输出两个字符串，标准输出和标准错误。

在 ClustalW 的例子中，当你使用程序时，所有重要的输出都被保存到输出文件中。所有打印在屏幕上的内容（通过 stdout or stderr）可以被忽略掉（假设它已经成功运行）。

当运行 ClustalW 的时候，我们所关心的往往是输出的序列比对文件和指导树文件。ClustalW 会自动根据输入数据的文件名来命名输出文件。在本例中，输出文件将是 opuntia.aln。当你成功运行完 ClustalW 以后，你可以使用 Bio.AlignIO 来读取输出结果：

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("opuntia.aln", "clustal")
>>> print align
SingleLetterAlphabet() alignment with 7 rows and 906 columns
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191
TATACATAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191
```

另一个输出文件 opuntia.dnd 中包含有一个 newick 格式的指导树，你可以使用 Biopython 中的 Bio.Phylo 来读取它：

```
>>> from Bio import Phylo
>>> tree = Phylo.read("opuntia.dnd", "newick")
>>> Phylo.draw_ascii(tree)
```

```

----- gi|6273291|gb|AF191665.1|AF191665
|
|----- gi|6273290|gb|AF191664.1|AF191664
|----- gi|6273289|gb|AF191663.1|AF191663
|
|----- gi|6273287|gb|AF191661.1|AF191661
|
|----- gi|6273286|gb|AF191660.1|AF191660
|
|----- gi|6273285|gb|AF191659.1|AF191659
|----- gi|6273284|gb|AF191658.1|AF191658

```

13 章中详细介绍了如何使用 Biopython 对进化树数据进行处理。

6.4.2 MUSCLE

MUSCLE 是另一个较新的序列比对工具，Biopython 的 Bio.Align.Applications 中也有针对 Muscle 的打包程序。与 ClustalW 一样，我们也建议你先在命令行界面下使用 MUSCLE 以后再使用 Biopython 打包程序。你会发现，Biopython 的打包程序非常严格地包括了所有命令行输入参数：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> help(MuscleCommandline)
...
```

作为最简单的例子，你只需要一个 Fasta 格式的数据文件作为输入。例如：`opuntia.fasta` 然后你可以告诉 MUSCLE 来读取该 FASTA 文件，并将序列比对写出：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.txt")
>>> print cline
muscle -in opuntia.fasta -out opuntia.txt
```

注意，MUSCLE 使用“-in”和“-out”来指定输入和输出文件，而在 Biopython 中，我们使用“input”和“out”作为关键字来指定输入输出。这是由于“in”是 Python 的一个关键词而被保留。

默认情况下，MUSCLE 的输出文件将是包含间隔 (gap) 的 FASTA 格式文件。当你指定 `format=fasta` 时，`Bio.AlignIO` 能够读取该 FASTA 文件。你也可以告诉 MUSCLE 来输出 ClustalW-like 的文件结果：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.aln", clw=True)
>>> print cline
muscle -in opuntia.fasta -out opuntia.aln -clw
```

或者，严格的 ClustalW 的输出文件（这将输出原始的 ClustalW 的文件标签）。例如：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> cline = MuscleCommandline(input="opuntia.fasta", out="opuntia.aln", clwstrict=True)
>>> print cline
muscle -in opuntia.fasta -out opuntia.aln -clwstrict
```

你可以使用 `Bio.AlignIO` 的 `format="clustal"` 参数来读取这些序列比对输出。

MUSCLE 也可以处理 GCG 和 MSF（使用 `msf` 参数）甚至 HTML 格式，但是目前 Biopython 并不能读取它们。

你也可以设置 MUSCLE 其它的可选参数，例如最大数目的迭代数。具体信息请查阅 Biopython 的内部帮助文档。

6.4.3 MUSCLE 标准输出

使用以上的 MUSCLE 命令行将会把序列比对结果写出到一个文件中。然而 MUSCLE 也允许你将序列比对结果作为系统的标准输出。Biopython 打包程序可以利用这一特性来避免创建一个临时文件。例如：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> print muscle_cline
muscle -in opuntia.fasta
```

如果你使用打包程序运行上述命令，程序将返回一个字符串对象。为了读取它，我们可以使用 `StringIO` 模块。记住 MUSCLE 将默认以 FASTA 格式输出序列比对：

```
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> stdout, stderr = muscle_cline()
>>> from StringIO import StringIO
>>> from Bio import AlignIO
>>> align = AlignIO.read(StringIO(stdout), "fasta")
>>> print align
```



```

SingleLetterAlphabet() alignment with 7 rows and 906 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191663
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191665
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191664
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191661
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191660
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191659
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191658

```

以上是一个非常简单的例子，如果你希望处理较大的输出数据，我们并不建议你将它们全部读入内存中。对于这种情况，`subprocess` 模块可以非常方便地处理。例如：

```

>>> import subprocess
>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(input="opuntia.fasta")
>>> child = subprocess.Popen(str(muscle_cline),
...                           stdout=subprocess.PIPE,
...                           stderr=subprocess.PIPE,
...                           shell=(sys.platform!="win32"))
>>> from Bio import AlignIO
>>> align = AlignIO.read(child.stdout, "fasta")
>>> print align
SingleLetterAlphabet() alignment with 7 rows and 906 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF191663
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273291|gb|AF191665.1|AF191665
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF191664
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF191661
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF191660
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF191659
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF191658

```

6.4.4 以标准输入和标准输出使用 MUSCLE

事实上，我们并不需要将序列放在一个文件里来使用 MUSCLE。MUSCLE 可以读取系统标准输入的内容。注意，这有一点高级和繁琐，若非必须，你可以不用关心这个技术。

为了让 MUSCLE 读取标准输入的内容，我们首先需要将未排列的序列以 `SeqRecord` 对象的形式读入到内存。在这里，我们将以一个规则来选择特定的序列（序列长度小于 900bp 的），使用生成器表达式。

```

>>> from Bio import SeqIO
>>> records = (r for r in SeqIO.parse("opuntia.fasta", "fasta") if len(r) < 900)

```

随后，我们需要建立一个 MUSCLE 命令行，但是不指定输入和输出（MUSCLE 默认为标准输入和标准输出）。这里，我们将指定输出格式为严格的 Clustal 格式：

```

>>> from Bio.Align.Applications import MuscleCommandline
>>> muscle_cline = MuscleCommandline(clwstrict=True)
>>> print muscle_cline
muscle -clwstrict

```

我们使用 Python 的内置模块 `subprocess` 来实现这一目的：

```

>>> import subprocess
>>> import sys
>>> child = subprocess.Popen(str(cline),
...                           stdin=subprocess.PIPE,
...                           stdout=subprocess.PIPE,

```

```
...         stderr=subprocess.PIPE,
...         shell=(sys.platform!="win32"))
```

这一命令将启动 MUSCLE，但是它将会等待 FASTA 格式的输入数据。我们可以通过标准输入句柄来提供给它：

```
>>> SeqIO.write(records, child.stdin, "fasta")
6
>>> child.stdin.close()
```

在将 6 条序列写入句柄后，MUSCLE 仍将会等待，判断是否所有的 FASTA 序列全部输入完毕了。我们可以关闭句柄来提示给 MUSCLE。这时，MUSCLE 将开始运行。最后，我们可以在标准输出中获得结果：

```
>>> from Bio import AlignIO
>>> align = AlignIO.read(child.stdout, "clustal")
>>> print align
SingleLetterAlphabet() alignment with 6 rows and 900 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF19166
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF19166
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF19165
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF19165
```

现在我们在没有创建一个 FASTA 文件的情况下获得了一个序列比对。然而，由于你没有在 Biopython 外运行 MUSCLE，这会使调试程序的难度增大，而且存在程序跨平台使用的问题（Windows 和 Linux）。

如果你觉得 subprocess 不方便使用，Biopython 提供了另一种方式。如果你用 `muscle_cline()` 来运行外部程序（如 MUSCLE），你可以用一个字符串对象作为输入。例如，你可以以这种方式使用：`muscle_cline(stdin=...)`。假如你的序列文件不大，你可以将其储存为 `StringIO` 对象（具体见 22.1）：

```
>>> from Bio import SeqIO
>>> records = (r for r in SeqIO.parse("opuntia.fasta", "fasta") if len(r) < 900)
>>> from StringIO import StringIO
>>> handle = StringIO()
>>> SeqIO.write(records, handle, "fasta")
6
>>> data = handle.getvalue()
```

你可以以下方式运行外部程序和读取结果：

```
>>> stdout, stderr = muscle_cline(stdin=data)
>>> from Bio import AlignIO
>>> align = AlignIO.read(StringIO(stdout), "clustal")
>>> print align
SingleLetterAlphabet() alignment with 6 rows and 900 columns
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273290|gb|AF191664.1|AF19166
TATACATTAAAGGAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273289|gb|AF191663.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273287|gb|AF191661.1|AF19166
TATACATAAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273286|gb|AF191660.1|AF19166
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273285|gb|AF191659.1|AF19165
TATACATTAAAGAAGGGGGATGCGGATAAATGGAAAGGCGAAAG...AGA gi|6273284|gb|AF191658.1|AF19165
```

你可能觉得这种方式更便捷，但它需要更多的内存（这是由于我们是以字符串对象来储存输入的 FASTA 文件和输出的 Clustal 排列）。

6.4.5 EMBOSS 包的序列比对工具——needle 和 water

EMBOSS 包有两个序列比对程序——water 和 needle 来实现 Smith-Waterman 做局部序列比对 (local alignment) 和 Needleman-Wunsch 算法来做全局排列 (global alignment)。这两个程序具有相同的使用方式，因此我们仅以 needle 为例。

假设你希望做全局的序列两两排列，你可以将 FASTA 格式序列以如下方式储存：

```
>HBA_HUMAN
MVLSPADKTNVKAAWGKVGAGHAGEYGAEALERMFSLFPTTKTYFPHFDLSHGSAQVKGHG
KKVADALTNVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTP
AVHASLDKFLASVSTVLTSKYR
```

以上内容在 alpha.fasta 文件中，另一个在 beta.fasta 中如下：

```
>HBB_HUMAN
MVHLTPEEKSAVTALWGKVNVDEVGGEALGRLLVVYPWTQRFFESFGDLSTPDAVMGNPK
VKAHGKKVLGAFSDGLAHLNKLKGTFAATLSEHCDKLVHPENFRLLGNVLVCVLAHHFG
KEFTTPVQAAAYQKVAVGAVANALAHKYH
```

让我们开始使用一个完整的 needle 命令行对象：

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline(asequence="alpha.faa", bsequence="beta.faa",
...                                   gapopen=10, gapextend=0.5, outfile="needle.txt")
>>> print needle_cline
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
```

你可能会疑问，为什么不直接在终端里运行这一程序呢？你会发现，它将进行一个序列两两间的排列，并把结果记录在 needle.txt 中（以 EMBOSS 默认的序列比对格式）。

即使你安装了 EMBOSS，使用以上命令仍可能会出错，你可能获得一个错误消息“command not found”，尤其是在 Windows 环境中。这很可能是由于 EMBOSS 工具的安装目录并不在系统的 PATH 中。遇到这种情况，你既可以更新系统的环境变量，也可以在 Biopython 中指定 EMBOSS 的安装路径。例如：

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline(r"C:\EMBOSS\needle.exe",
...                                   asequence="alpha.faa", bsequence="beta.faa",
...                                   gapopen=10, gapextend=0.5, outfile="needle.txt")
```

在 Python 中，\n 和 \t 分别意味着换行符和制表符。而在字符串前有一个“r”代表着 raw 字符串（\n 和 \t 将保持它们本来的状态）。

现在你可以自己尝试着手动运行 EMBOSS 工具箱中的程序，比较一下各个参数以及其对应的 Biopython 打包程序帮助文档：

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> help(NeedleCommandline)
...

```

提示：你也可以指定特定的参数设置。例如：

```
>>> from Bio.Emboss.Applications import NeedleCommandline
>>> needle_cline = NeedleCommandline()
>>> needle_cline.asequence="alpha.faa"
>>> needle_cline.bsequence="beta.faa"
>>> needle_cline.gapopen=10
>>> needle_cline.gapextend=0.5
>>> needle_cline.outfile="needle.txt"
>>> print needle_cline
```

```
needle -outfile=needle.txt -asequence=alpha.faa -bsequence=beta.faa -gapopen=10 -gapextend=0.5
>>> print needle_cline.outfile
needle.txt
```

现在我们获得了一个 `needle` 命令行，并希望在 Python 中运行它。我们在之前解释过，如果你希望完全地控制这一过程，`subprocess` 是最好的选择，但是如果你只是想尝试使用打包程序，以下命令足以达到目的：

```
>>> stdout, stderr = needle_cline()
>>> print stdout + stderr
Needleman-Wunsch global alignment of two sequences
```

随后，我们需要载入 `Bio.AlignIO` 模块来读取 `needle` 输出（`emboss` 格式）：

```
>>> from Bio import AlignIO
>>> align = AlignIO.read("needle.txt", "emboss")
>>> print align
SingleLetterAlphabet() alignment with 2 rows and 149 columns
MV-LSPADKTNVKAAWGKVGAGHAGEYGAEALERMFSLFPTTKTY...KYR HBA_HUMAN
MVHLTPEEKSAVTALWGKV--NVDEVGGEALGRLLVVYPWTQRF...KYH HBB_HUMAN
```

在这个例子中，我们让 `EMBOSS` 将结果保存到一个输出文件中，但是你也可以让其写入标准输出中（这往往是在不需要临时文件的情况下的选择，你可以使用 `stdout=True` 参数而不是 `outfile` 参数）。与 `MUSCLE` 的例子一样，你也可以从标准输入里读取序列（`asequence="stdin"` 参数）。

以上例子仅仅介绍了 `needle` 和 `water` 最简单的使用。一个有用的小技巧是，第二个序列文件可以包含有多个序列，`EMBOSS` 工具将每个序列与第一个文件进行两两序列比对。

注意，`Biopython` 有它自己的两两比对模块 `Bio.pairwise2`（用 C 语言编写）。但是它无法与序列比对对象一起工作，因此我们不在本章讨论它。具体信息请查阅模块的 `docstring`（内部帮助文档）。

第 7 章 BLAST

嗨，每个人都喜欢 BLAST，对吧？我是指，通过 BLAST 把你的序列和世界上已知的序列比较是多么简单方便啊。不过，这章当然不是讲 Blast 有多么酷，因为我们都已经知道了。这章是来解决使用 Blast 的一些麻烦地方——处理大量的 BLAST 比对结果数据通常是困难的，还有怎么自动运行 BLAST 序列比对。

幸运的是，Biopython 社区的人早就了解了这些难处。所以，他们已经发展了很多工具来简化 BLAST 使用和结果处理。这章会具体讲解怎么用这些工具来做些有用的事情。

使用 BLAST 通常可以分成 2 个步。这两步都可以用上 Biopython。第一步，提交你的查询序列，运行 BLAST，并得到输出数据。第二步，用 Python 解析 BLAST 的输出，并作进一步分析。

你第一次接触并运行 BLAST 也许就是通过 NCBI 的 web 服务。事实上，你可以通过多种方式 (这些方式可以分成几类) 来使用 BLAST。这些方式最重要的区别在于你是在你的自己电脑上运行一个本地 BLAST，还是在远程服务器 (另外一台电脑，通常是 NCBI 的服务器) 上运行。我们将在一个 Python 脚本里调用 NCBI 在线 BLAST 服务来开始这章的内容。

注意: 接下来的第 8 章介绍的 Bio.SearchIO 是一个 Biopython 实验性质的模块。我们准备最终用它来替换原来的 Bio.Blast 模块。因为它提供了一个更为通用的序列搜索相关的框架。不过，在这个模块的稳定版本发布之前，在实际工作中的代码里，请继续用 Bio.Blast 模块来处理 NCBI BLAST。

7.1 通过 Internet 运行 BLAST

我们使用 Bio.Blast.NCBIWWW 模块的函数 qblast() 来调用在线版本的 BLAST。这个函数有 3 个必需的参数:

- 第一个参数是用来搜索的 blast 程序，这是小写的字符串。对这个参数的选项和描述可以在 http://www.ncbi.nlm.nih.gov/BLAST/blast_program.shtml 查到。目前 qblast 只支持 blastn, blastp, blastx, tblast 和 tblastx。
- 第二个参数指定了将要搜索的数据库。同样地，这个参数的选项也可以在 http://www.ncbi.nlm.nih.gov/BLAST/blast_databases.shtml 查到
- 第三个参数是一个包含你要查询序列的字符串。这个字符串可以是序列的本身 (fasta 格式的)，或者是一个类似 GI 的 id。

qblast 函数还可以接受许多其他选项和参数。这些参数基本上类似于你在 BLAST 网站页面能够设置的参数。在这里我们只强调其中的一些：

- qblast 函数可以返回多种格式的 BLAST 结果。你可以通过可选参数 format_type 指定格式关键字为："HTML", "Text", "ASN.1", 或 "XML"。默认格式是 "XML"，这是解析器期望的格式，7.3 节对其有详细的描述。
- 参数 expect 指定期望值，即阈值 e-value。

更多可选的 BLAST 参数，请参照 NCBI 的文档，或者是 Biopython 内置的文档。

```
>>> from Bio.Blast import NCBIWWW
>>> help(NCBIWWW.qblast)
...
```

请注意，NCBI BLAST 网站上的默认参数和 QBLAST 的默认参数不完全相同。如果你得到了不同的结果，你就需要检查下参数设置（比如，e-value 阈值和 gap 值）。

举个例子，如果你有条核酸序列，想使用 BLAST 对核酸数据库（nt）进行搜索，已知这条查询序列的 GI 号，你可以这样做：

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

或者，我们想要查询的序列在 FASTA 文件中，那么我们只需打开这个文件并把这条记录读入到字符串，然后用这个字符串作为查询参数：

```
>>> from Bio.Blast import NCBIWWW
>>> fasta_string = open("m_cold.fasta").read()
>>> result_handle = NCBIWWW.qblast("blastn", "nt", fasta_string)
```

我们同样可以读取 FASTA 文件为一个 SeqRecord 序列对象，然后以这个序列自身作为参数：

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.seq)
```

只提供序列意味着 BLAST 会自动分配给你一个 ID。你可能更喜欢用 SeqRecord 对象的 format 方法来包装一个 fasta 字符串，因为这个对象会包含 fasta 文件中已有的 ID

```
>>> from Bio.Blast import NCBIWWW
>>> from Bio import SeqIO
>>> record = SeqIO.read("m_cold.fasta", format="fasta")
>>> result_handle = NCBIWWW.qblast("blastn", "nt", record.format("fasta"))
```

如果你的序列在一个非 FASTA 格式的文件中并且你用 Bio.SeqIO（看第5章）把序列取出来了，那么这个方法更有用。

不论你给 qblast() 函数提供了什么参数，都应该返回一个 handle object 的结果（默认是 XML 格式）。下一步就是将这个 XML 输出解析为代表 BLAST 搜索结果的 Python 对象（7.3）。不过，也许你想先把这个 XML 输出保存一个本地文件副本。当调试从 BLAST 结果提取信息的代码的时候，我发现这样做尤其有用。（因为重新运行在线 BLAST 搜索很慢并且会浪费 NCBI 服务器的运行时间）。

这里我们需要注意下：因为用 result_handle.read() 来读取 BLAST 结果只能用一次 - 再次调用 result_handle.read() 会返回一个空的字符串。

```
>>> save_file = open("my_blast.xml", "w")
>>> save_file.write(result_handle.read())
>>> save_file.close()
>>> result_handle.close()
```

这些做好后，结果已经存储在 my_blast.xml 文件中了并且原先的 handle 中的数据已经被全部提取出来了（所以我们把它关闭了）。但是，BLAST 解析器的 parse 函数（描述见 7.3）采用一个文件句柄类的对象，所以我们只需打开已经保存的文件作为输入。

```
>>> result_handle = open("my_blast.xml")
```

既然现在已经把 BLAST 的结果又一次读回 handle，我们可以分析下这些结果。所以我们正好可以去读关于结果解析的章节（看下面 7.3）。你现在也许想跳过去看看吧...

7.2 本地运行 BLAST

7.2.1 介绍

在本地运行 BLAST (跟通过 internet 运行比, 见 7.1) 至少有 2 个主要优点:

- 本地运行 BLAST 可能比通过 internet 运行更快;
- 本地运行可以让你建立自己的数据库来对序列进行搜索。

处理有版权的或者没有发表的序列数据也许是本地运行 BLAST 的另一个原因。你也许不能泄露这些序列数据, 所以没法提交给 NCBI 来 BLAST。

不幸的是, 本地运行也有些缺点 - 安装所有的东东并成功运行需要花些力气:

- 本地运行 BLAST 需要你安装相关命令行工具。
- 本地运行 BLAST 需要安装一个很大的 BLAST 的数据库 (并且需要保持数据更新)。

更令人困惑的是, 至少有 4 种不同的 BLAST 安装程序包, 并且还有其他的一些工具能产生类似的 BLAST 输出文件, 比如 BLAT。

7.2.2 单机版的 NCBI 老版本 BLAST

NCBI “老版本”BLAST 包括命令行工具 `blastall`, `blastpgp` 和 `rpsblast`。这是 NCBI 发布它的替代品 BLAST+ 前使用最为广泛的单机版 BLAST 工具。

Bio.Blast.Applications 模块有个对老版本 NCBI BLAST 工具像 `blastall`, `blastpgp` 和 `rpsblast` 的封装, 并且在 Bio.Blast.NCBIStandalone 还有个辅助函数。这些东东现在都被认为是过时的, 并且当用户们迁移到 BLAST+ 程序套件后, 这些都会被弃用, 最终从 Biopython 删除。

为了减少你的困惑, 我们在这个指南中不会提到怎么从 Biopython 调用这些老版本的工具。如果你有兴趣, 可以看下在 Biopython 1.52 中包含的基本指南。(看下 `biopython-1.52.tar.gz` 或者 `biopython-1.52.zip` 中 Doc 目录下的指南的 PDF 文件或者 HTML 文件)。

7.2.3 单机版 NCBI BLAST+

NCBI “新版本”的 BLAST+ 在 2009 年发布。它替代了原来老版本的 BLAST 程序包。Bio.Blast.Applications 模块包装了这些新工具像 `blastn`, `blastp`, `blastx`, `tblastn`, `tblastx` (这些以前都是由 `blastall` 处理)。而 `rpsblast` 和 `rpstblastn` (替代了原来的 `rpsblast`)。我们这里不包括对 `makeblastdb` 的包装, 它在 BLAST+ 中用于从 FASTA 文件建立一个本地 BLAST 数据库, 还有其在老版本 BLAST 中的等效工具 `formatdb`。

这节将简要地介绍怎样在 Python 中使用这些工具。如果你已经阅读了并试过 6.4 节的序列联配 (alignment) 工具, 下面介绍的方法应该是很简单直接的。首先, 我们构建一个命令行字符串 (就像你使用单机版 BLAST 的时候, 在终端打入命令行一样)。然后, 我们在 Python 中运行这个命令。

举个例子, 你有个 FASTA 格式的核酸序列文件, 你想用它通过 BLASTX (翻译) 来搜索非冗余 (NR) 蛋白质数据库。如果你 (或者你的系统管理员) 下载并安装好了这个数据库, 那么你只要运行:

```
blastx -query opuntia.fasta -db nr -out opuntia.xml -evalue 0.001 -outfmt 5
```

这样就完成了运行 BLASTX 查找非冗余蛋白质数据库, 用 0.001 的 e 值并产生 XML 格式的输出结果文件 (这样我们可以继续下一步解析)。在我的电脑上运行这条命令花了大约 6 分钟 - 这就是为什么我们需要保存输出到文件。这样我们就可以在需要时重复任何基于这个输出的分析。

在 Biopython 中，我们可以用 NCBI BLASTX 包装模块 `Bio.Blast.Applications` 来构建命令行字符串并运行它：

```
>>> from Bio.Blast.Applications import NcbiblastxCommandline
>>> help(NcbiblastxCommandline)
...
>>> blastx_cline = NcbiblastxCommandline(query="opuntia.fasta", db="nr", evalue=0.001,
...                                       outfmt=5, out="opuntia.xml")
>>> blastx_cline
NcbiblastxCommandline(cmd='blastx', out='opuntia.xml', outfmt=5, query='opuntia.fasta',
db='nr', evalue=0.001)
>>> print blastx_cline
blastx -out opuntia.xml -outfmt 5 -query opuntia.fasta -db nr -evalue 0.001
>>> stdout, stderr = blastx_cline()
```

在这个例子中，终端里应该没有任何从 BLASTX 的输出，所以 `stdout` 和 `stderr` 是空的。你可能想要检查下输出文件 `opuntia.xml` 是否已经创建。

如果你回想下这个指南的中的早先的例子，`opuntia.fasta` 包含 7 条序列，所以 BLAST XML 格式的结果输出文件应该包括多个结果。因此，我们在下面的 7.3 节将用 `Bio.Blast.NCBIXML.parse()` 来解析这个结果文件。

7.2.4 WU-BLAST 和 AB-BLAST

你也许会碰到 Washington University BLAST (WU-BLAST)，和它的后继版本 ‘Advanced Biocomputing BLAST’ <<http://blast.advbiocomp.com>> ‘_ (AB-BLAST, 在 2009 年发布，免费但是没有开源)。这些程序包包括了命令工具行 `wu-blastall` 和 `ab-blastall`。

Biopython 目前还没有提供调用这些工具的包装程序，但是应该可以解析它们与 NCBI 兼容的输出结果。

7.3 解析 BLAST 输出

就像上面提过的那样，BLAST 能生成多种格式的输出，比如 XML，HTML 和纯文本格式。以前，Biopython 有针对 HTML 和纯文本格式输出文件的解析器，因为当时只有这两种格式的输出结果文件。不幸的是，这两种方式的 BLAST 输出结果一直在变动，而每次变动就会导致解析器失效。所以，我们删除了针对 HTML 格式的解析器，不过纯文本格式的解析还可以用（见 7.5）。使用这个解析器有一定的风险，它可能工作也可能无效，依赖于你正在使用哪个 BLAST 版本。

跟上 BLAST 输出文件格式的改变很难，特别是当用户使用不同版本的 BLAST 的时候。我们推荐使用 XML 格式的解析器。因为最近版本的 BLAST 能生成这种格式的输出结果。XML 格式的输出不仅比 HTML 和纯文本格式的更稳定，而且解析起来更加容易自动化，从而提高整个 Biopython 整体的稳定性。

你可以通过好几个途径来获得 XML 格式的 BLAST 输出文件。对解析器来说，不管你是怎么生成输出的，只要是输出的格式是 XML 就行。

- 你可以通过 Biopython 来运行因特网上的 BLAST，就像 7.1 节描述的那样。
- 你可以通过 Biopython 来运行本地的 BLAST，就像 7.2 节描述的那样。
- 你可以在通过浏览器在 NCBI 网站上进行 BLAST 搜索，然后保存结果文件。你需要选择输出结果文件是 XML 格式的，并保存最终的结果网页（你知道，就是包含所有有趣结果的那个网页）到文件。
- 你也可以直接运行本地电脑上的 BLAST，不通过 Biopython，保存输出结果到文件。同样的你也需要选择输出文件格式为 XML。

关键点就是你不必用 Biopython 脚本来获取数据才能解析它。通过以上任何一种方式获取了结果输出，你然后需要获得文件句柄来处理它。在 Python 中，一个文件句柄就是一种用于描述到任何信息源的输入的良好通用的方式，以便于这些信息能够使用 `read()` 和 `readline()` 函数（见章节 22.1）来获取。

如果你一直跟着上几节用来和 BLAST 交互的代码的话，你已经有了个 `result_handle`，一个用来得到 BLAST 的结果文件句柄。比如通过 GI 号来进行一个在线 BLAST 搜索：

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nt", "8332116")
```

如果你通过其他方式运行了 BLAST，并且 XML 格式的 BLAST 结果输出文件是 `my_blast.xml`，那么你只需要打开文件来读：

```
>>> result_handle = open("my_blast.xml")
```

好的，现在我们已经有了个文件句柄，可以解析输出结果了。解析结果的代码很短。如果你想要一条 BLAST 输出结果（就是说，你只用了一条序列去搜索）：

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

或者，你有许多搜索结果（就是说，你用了多条序列去 BLAST 搜索）

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
```

就像 `Bio.SeqIO` 和 `Bio.AlignIO`（参见第 5 和第 6 章），我们有一对输入函数，`read` 和 `parse`。当你只有一个输出结果的时候用 `read`。当你有许多输出结果的时候，可以用 `parse` 这个迭代器。但是，我们调用函数获得结果不是 `SeqRecord` 或者 `MultipleSeqAlignment` 对象，我们得到 BLAST 记录对象。

为了能处理 BLAST 结果文件很大有很多结果这种情况，`NCBIXML.parse()` 返回一个迭代器。简单来说，一个迭代器可以让你一个接着一个地获得 BLAST 的搜索结果。

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
# ... do something with blast_record
>>> blast_record = blast_records.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
# No further records
```

或者，你也可以使用 `for` - 循环

```
>>> for blast_record in blast_records:
...     # Do something with blast_record
```

注意对每个 BLAST 搜索结果只能迭代一次。通常，对于每个 BLAST 记录，你可能会保存你感兴趣的信息。如果你想保存所有返回的 BLAST 记录，你可以把迭代转换成列表。

```
>>> blast_records = list(blast_records)
```

现在，你可以像通常的做法通过索引从这个列表中获得每一条 BLAST 结果。如果你的 BLAST 输出结果文件很大，那么当把它们全部放入一个列表时，你也许会遇到内存不够的情况。

一般来说，你会一次运行一个 BLAST 搜索。然后，你只需提取第一条 BLAST 搜索记录到 `blast_records`：

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
>>> blast_record = blast_records.next()
```

or more elegantly:

或者更加优雅地：

```
>>> from Bio.Blast import NCBIXML
>>> blast_record = NCBIXML.read(result_handle)
```

我猜你现在在想 BLAST 搜索记录中到底有什么。

7.4 BLAST 记录类

一个 BLAST 搜索结果记录包括了所有你想要从中提取出来的信息。现在，我们将用一个例子说明你怎么从 BLAST 搜索结果提取出一些信息。但是，如果你想从 BLAST 搜索结果获得的信息没有在这里提到，你可以详细阅读 BLAST 搜索记录类，并且可以参考下源代码或者是自动生成的文档 - 文档字符串里面包含了许多关于各部分源代码是什么的很有用的信息。

继续我们的例子，让我们打印出所有大于某一特定阈值的 BLAST 命中结果的一些汇总信息。代码如下：

```
>>> E_VALUE_THRESH = 0.04

>>> for alignment in blast_record.alignments:
...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
...             print '****Alignment****'
...             print 'sequence:', alignment.title
...             print 'length:', alignment.length
...             print 'e value:', hsp.expect
...             print hsp.query[0:75] + '...'
...             print hsp.match[0:75] + '...'
...             print hsp.sbjct[0:75] + '...'
```

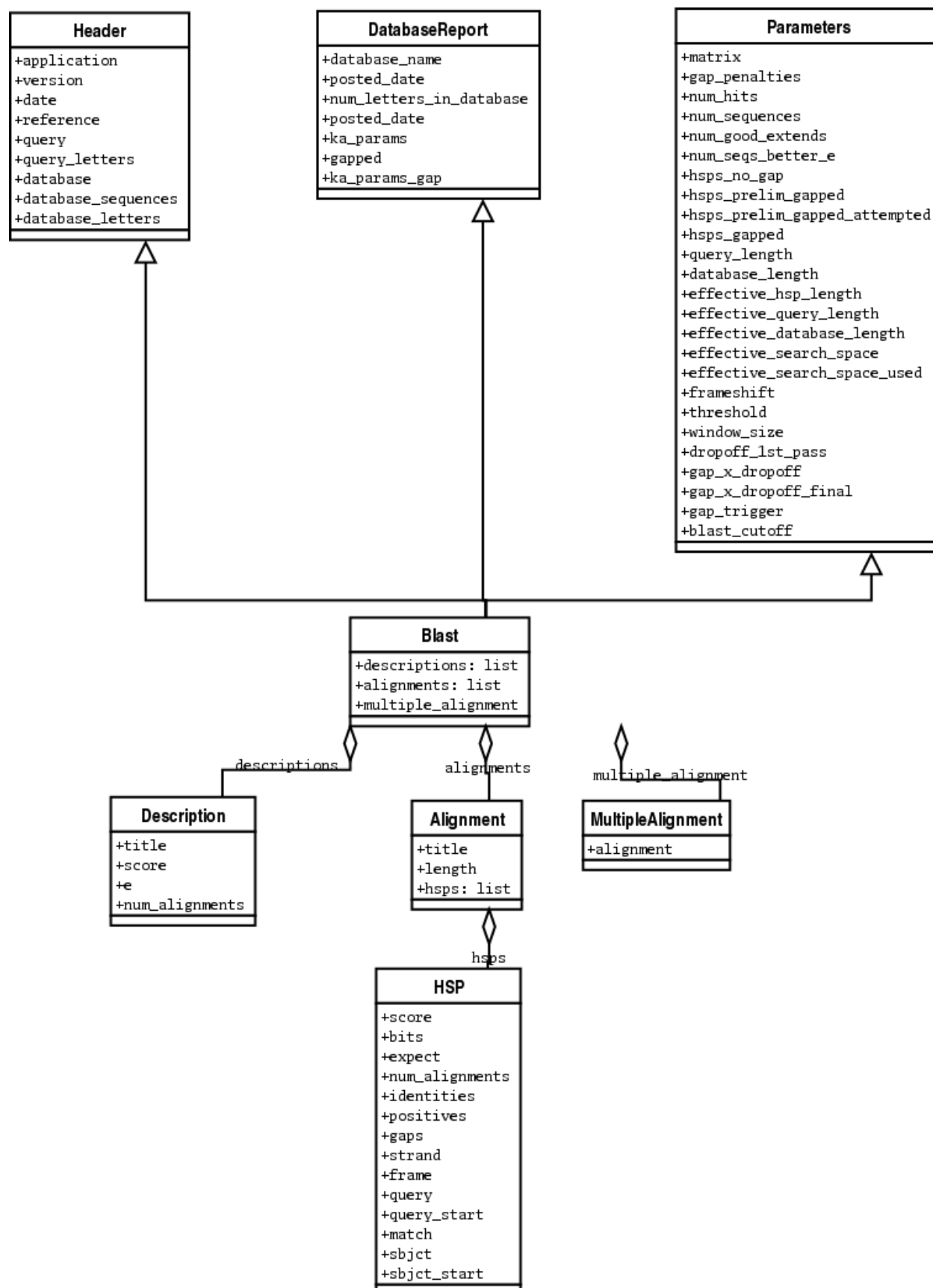
上面代码会打印出如下图的总结报告：

```
****Alignment****
sequence: >gb|AF283004.1|AF283004 Arabidopsis thaliana cold acclimation protein WCOR413-like protein
alpha form mRNA, complete cds
length: 783
e value: 0.034
tacttgttgatattggatcgaaactggagaaccaacatgctcacgtcacttttagtccttacatattcctc...
||||||| | ||||| || ||| || || ||||| ||||| | | ||||| ||| ||...
tacttgttggtgttgatcgaaactggagaaccaatggaagacgaatatgctcacatcacttctcattccttacatcttctc...
```

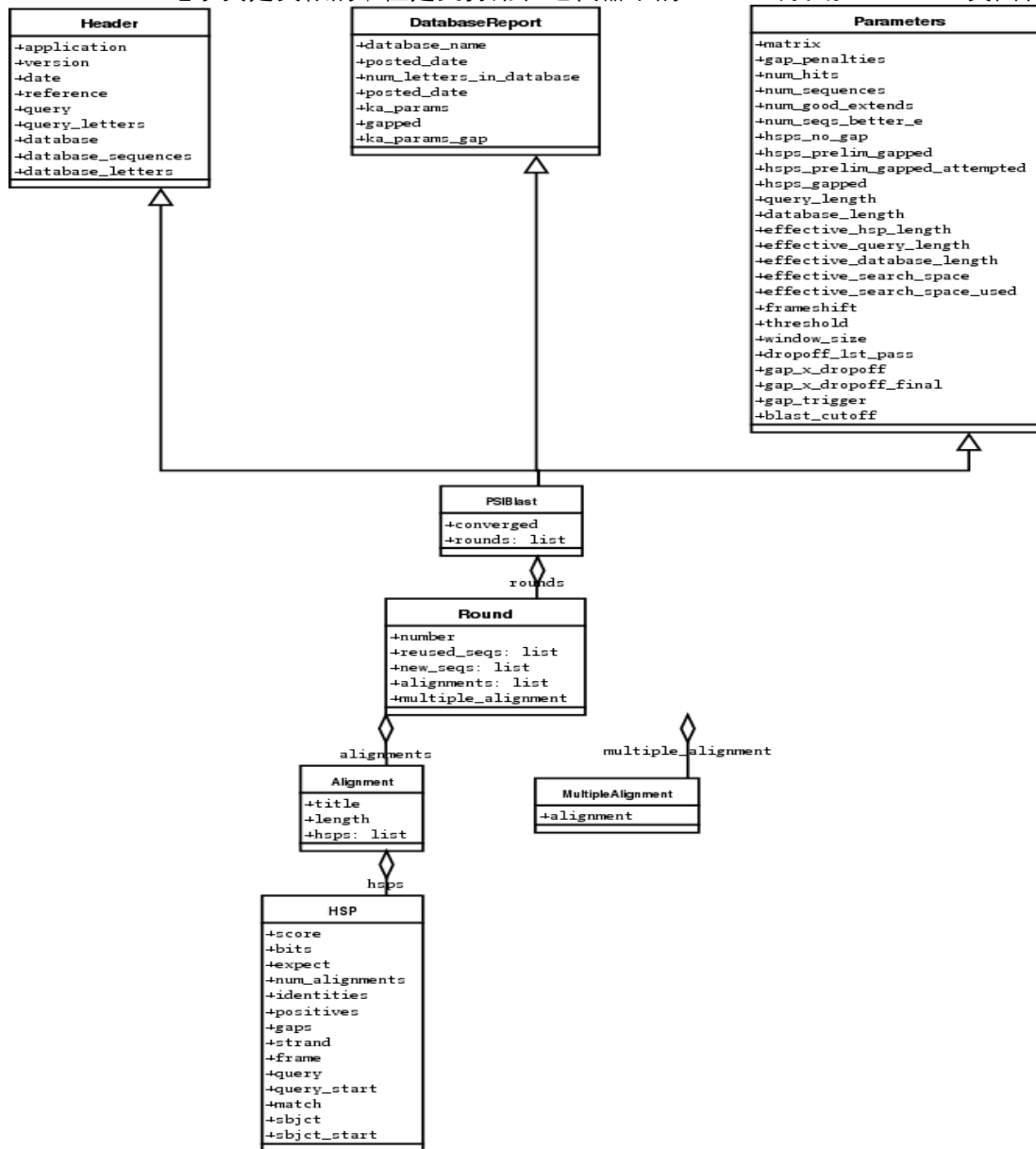
基本上，一旦你解析了 BLAST 搜索结果文件，你可以提取任何你需要的信息。当然，这取决于你想要获得什么信息。但是希望这里的例子能够帮助你开始工作。

在用 Biopython 提取 BLAST 搜索结果信息的时候，重要的是你需要考虑到信息存储在什么 (Biopython) 对象中。在 Biopython 中，解析器返回 `Record` 对象，这个对象可以是 `Blast` 类型的，也可以是 `PSIBlast` 类型的，具体哪个取决你解析什么。这些对象的定义都可以在 `Bio.Blast.Record` 找到并且很完整。

下面是我尝试画的 Blast 和 PSIBlast 记录类的 UML 图。如果你对 UML 图很熟悉，不妨看看下面的 UML 图是否有错误或者可以改进的地方，如果有，请联系我。BLAST 类图在这里[7.4](#)。



PSIBlast 记录类是类似的，但是支持用在迭代器中的 rounds 方法。PSIBlast 类图在这里 [7.4](#)。



7.5 废弃的 BLAST 解析器

老版本的 Biopython 有针对纯文本和 HTML 格式输出结果的解析器。但是经过几年我们发现维护这些解析器很困难。基本上，任何 BLAST 输出的任何小改变都会导致这些解析器失效。所以我们推荐你解析 XML 格式的 BLAST 输出结果，就像在 [7.3](#) 描述的那样。

取决于你使用 Biopython 的版本，纯文本格式的解析器也许有效也许失效。用这个解析器的所带来的风险由你自己承担。

7.5.1 解析纯文本格式的 BLAST 输出

纯文本格式的解析器在 `Bio.Blast.NCBISTandalone` 。

和 xml 解析器类似，我们也需要一个能够传给解析器的文件句柄。这个文件句柄必须实现了 `readline()` 方法。通常要获得这样文件句柄，既可以用 Biopython 提供的 `blastall` 或 `blastpgp` 函数来调用本地的 BLAST，或者从命令行运行本地的 BLAST，并且如下处理：

```
>>> result_handle = open("my_file_of_blast_output.txt")
```

好了，既然现在得到了个文件句柄（我们称它是 `result_handle` ），我们已经做好了解析它的准备。按下面的代码来解析：

```
>>> from Bio.Blast import NCBISTandalone
>>> blast_parser = NCBISTandalone.BlastParser()
>>> blast_record = blast_parser.parse(result_handle)
```

这样就能把 BLAST 的搜索结果报告解析到 Blast 记录类中（取决于你解析的对象，解析结果可能返回一条 Blast 或者 PSIBlast 记录）。这样你就可以从中提取信息了。在我们的例子里，我们来打印出大于某个阈值的所有比对的一个总结信息。

```
>>> E_VALUE_THRESH = 0.04
>>> for alignment in blast_record.alignments:
...     for hsp in alignment.hsps:
...         if hsp.expect < E_VALUE_THRESH:
...             print '****Alignment****'
...             print 'sequence:', alignment.title
...             print 'length:', alignment.length
...             print 'e value:', hsp.expect
...             print hsp.query[0:75] + '...'
...             print hsp.match[0:75] + '...'
...             print hsp.sbjct[0:75] + '...'
```

如果你已经读过 7.3 节关于解析 XML 格式的部分，你将会发现上面的代码和那个章节的是一样的。一旦你把输出文件解析到记录类中，你就能处理信息，不管你原来的 BLAST 输出格式是什么。很赞吧。

好，解析一条记录是不错，那么如果我有一个包含许多记录的 BLAST 文件 - 我该怎么处理它们呢？好吧，不要害怕，答案就在下个章节中。

7.5.2 解析包含多次 BLAST 结果的纯文本 BLAST 文件

我们可以用 BLAST 迭代器解析多次结果。为了得到一个迭代器，我们首先需要创建一个解析器，来解析 BLAST 的搜索结果报告为 Blast 记录对象。

```
>>> from Bio.Blast import NCBISTandalone
>>> blast_parser = NCBISTandalone.BlastParser()
```

然后，我们假定我们有一个连接到一大堆 blast 记录的文件句柄，我们把这个文件句柄叫做 `result_handle` 。怎么得到一个文件句柄在上面 blast 解析章节有详细描述。

好了，我们现在有了一个解析器和一个文件句柄，我们可以用以下命令来创建一个迭代器。

```
>>> blast_iterator = NCBISTandalone.Iterator(result_handle, blast_parser)
```

第二个参数，解析器，是可选的。如果我们没有提供一个解析器，那么迭代器将会一次返回一个原始的 BLAST 搜索结果。

现在我们已经有了个迭代器，就可以开始通过 `next()` 方法来获取 BLAST 记录（由我们的解析器产生）。

```
>>> blast_record = blast_iterator.next()
```

每次调用 `next` 都会返回一条我们能处理的新记录。现在我们可以遍历所有记录，并打印一个我们最爱、漂亮的、简洁的 BLAST 记录报告。

```
>>> for blast_record in blast_iterator:
...     E_VALUE_THRESH = 0.04
...     for alignment in blast_record.alignments:
...         for hsp in alignment.hsps:
...             if hsp.expect < E_VALUE_THRESH:
...                 print '****Alignment****'
...                 print 'sequence:', alignment.title
...                 print 'length:', alignment.length
...                 print 'e value:', hsp.expect
...                 if len(hsp.query) > 75:
...                     dots = '...'
...                 else:
...                     dots = ''
...                 print hsp.query[0:75] + dots
...                 print hsp.match[0:75] + dots
...                 print hsp.sbjct[0:75] + dots
```

迭代器允许你处理很多 blast 记录而不出现内存不足的问题。因为，它使一次处理一个记录。我曾经用大处理过一个非常巨大的文件，没有出过任何问题。

7.5.3 在巨大的 BLAST 纯文本文件中发现不对的记录

当我开始解析一个巨大的 blast 文件，有时候会碰到一个郁闷的问题就是解析器以一个 `ValueError` 异常终止了。这是个严肃的问题。因为你无法分辨导致 `ValueError` 异常的是解析器的问题还是 BLAST 的问题。更加糟糕是，你不知道在哪一行解析器失效了。所以，你不能忽略这个错误。不然，可能会忽视一个重要的数据。

我们以前必须写一些小脚本来解决这个问题。不过，现在 `Bio.Blast` 模块包含了 `BlastErrorParser`，可以更加简单地来解决这个问题。`BlastErrorParser` 和常规的 `BlastParser` 类似，但是它加了特别一层来捕获由解析器产生的 `ValueErrors` 异常，并尝试来诊断这些错误。

让我们来看看怎样用这个解析器 - 首先我们定义我们准备解析的文件和报告错误情况的输出文件。

```
>>> import os
>>> blast_file = os.path.join(os.getcwd(), "blast_out", "big_blast.out")
>>> error_file = os.path.join(os.getcwd(), "blast_out", "big_blast.problems")
```

现在我们想要一个 `BlastErrorParser`：

```
>>> from Bio.Blast import NCBIStandalone
>>> error_handle = open(error_file, "w")
>>> blast_error_parser = NCBIStandalone.BlastErrorParser(error_handle)
```

注意，解析器有个关于文件句柄的可选参数。如果传递了这个参数，那么解析器就会把产生 `ValueError` 异常的记录写到这个文件句柄中。不然的话，这些错误记录就不会被记录下来。

现在，我们可以像用常规的 blast 解析器一样地用 `BlastErrorParser`。特别的是，我们也许想要一个一次读入一个记录的迭代器并用 `BlastErrorParser` 来解析它。

```
>>> result_handle = open(blast_file)
>>> iterator = NCBIStandalone.Iterator(result_handle, blast_error_parser)
```

我们可以一次读一个记录，并且我们现在可以捕获并处理那些因为 Blast 引起的、不是解析器本身导致的错误。

```
>>> try:
...     next_record = iterator.next()
... except NCBIStandalone.LowQualityBlastError, info:
...     print "LowQualityBlastError detected in id %s" % info[1]
```

.next() 方法通常被 for 循环间接地调用。现在，BlastErrorParser 能够捕获如下的错误：

- ValueError - 这就是和常规 BlastParser 产生的一样的错误。这个错误产生是因为解析器不能解析某个文件。通常是因为解析器有 bug，或者是因为你使用解析器的版本和你 BLAST 命令的版本不一致。
- LowQualityBlastError - 当 Blast 一条低质量的序列时（比如，一条只有 1 个核苷酸的短序列），似乎 Blast 会终止并屏蔽掉整个序列，所有就没有什么可以解析了。这种情况下，Blast 就会产生一个不完整的报告导致解析器出现 ValueError 错误。LowQualityBlastError 错误在这种情况下产生。这个错误返回如下信息：
 - item[0] – The error message
 - item[0] - 错误消息
 - item[1] – The id of the input record that caused the error. This is really useful if you want to record all of the records that are causing problems.
 - item[1] - 导致错误产生的输入记录 id。如果你想记录所有导致问题记录的时候很有用。

就像上面提到的那样，BlastErrorParser 将会把有问题的记录写到指定的“error_handle”。然后，你可以排查这些有问题记录。你可以针对某条记录来调试解析器，或者找到你运行 blast 中的问题。无论哪种方式，这些都是有用的经验。

希望 BlastErrorParser 能帮你更简单的调试和处理一些数据巨大的 Blast 文件。

7.6 处理 PSI-BLAST

你可以通过 Bio.Blast.Applications 模块中的包装函数来运行单机版本的 PSI-BLAST（老版本的 NCBI 命令工具行 blastpgp 或者它的替代程序 psiblast）。

在写这篇指南的时候，没有迹象表明 NCBI 将会支持通过 internet 来进行 PSI-BLAST 搜索。

请注意 Bio.Blast.NCBIXML 解析器能读入并解析当前版本 PSI-BLAST 的、XML 格式的输出，但是像哪条序列在每个迭代循环中是新的还是复用的信息在 XML 格式输出中是没有的。如果，你需要这些信息你应该用纯文本输出和 Bio.Blast.NCBIStandalone 模块的 PSIBlastParser。

7.7 处理 RPS-BLAST

你可以通过 Bio.Blast.Applications 模块中的包装函数来运行单机版本的 RPS-BLAST（或者老版本的 NCBI 命令工具行 rpsblast 或者同样名字的替代程序）。

在写这篇指南的时候，没有迹象表明 NCBI 将会支持通过 internet 来进行 RPS-BLAST 搜索

你可以通过 Bio.Blast.NCBIXML 这个解析器来读入并解析当前版本的 RPS-BLAST 的 XML 格式的输出。

第 8 章 BLAST 和其他序列搜索工具 (实验性质的代码)

WARNING: 这章教程介绍了 Biopython 中一个 实验的 模块。它正在被加入到 Biopython 中，并且以一个预尾期的状态整理到教程当中，这样在我们发布稳定版的之前可以收到一系列的反馈和并作改进。那时有些细节可能会改变，并且用到当前 Bio.SearchIO 模块的脚本也需要更新。切记！为了与 NCBI BLAST 有关的代码可以稳定工作，请继续使用第 7 章介绍的 Bio.Blast。

生物序列的鉴定是生物信息工作的主要部分。有几个工具像 BLAST (可能是最流行的), FASTA , HMMER 还有许多其它的都有这个功能，每个工具都有独特的算法和途径。一般来说，这些工具都是用你的序列在数据库中搜索可能的匹配。随着序列数量的增加 (匹配数也会随之增加)，将会有成百上千的可能匹配，解析这些结果无疑变得越来越困难。理所当然，人工解析搜索结果变得不可能。而且你可能会同时用几种不同的搜索工具，每种工具都有独特的统计方法、规则和输出格式。可以想象，同时用多种工具搜索多条序列是多么恐怖的事。

我们对此非常了解，所以我们在 Biopython 构建了 Bio.SearchIO 亚模块。Bio.SearchIO 模块使从搜索结果中提取信息变得简单，并且可以处理不同工具的不同标准和规则。SearchIO 和 BioPerl 中模块名称一致。

在本章中，我们将学习 Bio.SearchIO 的主要功能，了解它可以做什么。我们将使用两个主要的搜索工具：BLAST 和 FASTA。它们只是用来阐明思路，你可以轻易地把工作流程应用到 Bio.SearchIO 支持的其他工具中。欢迎你使用我们将要用到的搜索结果文件。BLAST 搜索结果文件可以在 [这里](#) 下载。BLAT 输出结果文件可以在 [这里](#) 下载。两个结果文件都是用下面这条序列搜索产生的：

```
>mystery_seq
CCCTCTACAGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTTAGAGGG
```

BLAST 的 XML 结果是用 blastn 搜索 NCBI 的 refseq_rna 数据库得到的。对于 BLAT，数据库是 2009 年 2 月份的 hg19 人类基因组草图，输出格式是 PSL。

我们从 Bio.SearchIO 的对象模型的介绍开始。这个模型代表你的搜索结果，因此它是 Bio.SearchIO 的核心。然后，我们会介绍 Bio.SearchIO 常用的主要功能。

现在一切就绪，让我们开始第一步：介绍核心对象模型。

8.1 SearchIO 对象模型

尽管多数搜索工具的输出风格极为不同，但是它们蕴含的理念很相似：

- 输出文件可能包含一条或更多的搜索查询的结果。
- 在每次查询中，你会在给定的数据库中得到一个或更多的 hit (命中)。
- 在每个 hit 中，你会得到一个或更多包含 query (查询) 序列和数据库序列实际比对的区域。
- 一些工具如 BLAT 和 Exonerate 可能会把这些区域分成几个比对片段 (或在 BLAT 中称为区块，在 Exonerate 中称为可能外显子)。这并不是很常见，像 BLAST 和 HMMER 就不这么做。

知道这些共性之后，我们决定把它们作为创造 Bio.SearchIO 对象模型的基础。对象模型是 Python 对象组成的嵌套分级系统，每个对象都代表一个上面列出的概念。这些对象是：

- QueryResult，代表单个搜索查询。
- Hit，代表单个的数据库 hit。Hit 对象包含在 QueryResult 中，每个 QueryResult 中有 0 个或多个 Hit 对象。
- HSP (high-scoring pair (高分片段))，代表 query 和 hit 序列中有意义比对的区域。HSP 对象包含在 Hit 对象中，而且每个 Hit 有一个或更多的 HSP 对象。
- HSPFragment，代表 query 和 hit 序列中单个的邻近比对。HSPFragment 对象包含在 HSP 对象中。多数的搜索工具如 BLAST 和 HMMER 把 HSP 和 HSPFragment 合并，因为一个 HSP 只含有一个 HSPFragment。但是像 BLAT 和 Exonerate 会产生含有多个 HSPFragment 的 HSP。似乎有些困惑？不要紧，稍后我们将详细介绍这两个对象。

这四个对象是当你用 Bio.SearchIO 会碰到的。Bio.SearchIO 四个主要方法：read，parse，index 或 index.db 中任意一个都可以产生这四个对象。这些方法的会在后面的部分详细介绍。这部分只会用到 read 和 parse，这两个方法和 Bio.SeqIO 以及 Bio.AlignIO 中的 read 和 parse 方法功能相似：

- read 用于单 query 对输出文件进行搜索并且返回一个 QueryResult 对象。
- parse 用于多 query 对输出文件进行搜索并且返回一个可以 yield QueryResult 对象的 generator。

完成这些之后，让我们开始学习每个 Bio.SearchIO 对象，从 QueryResult 开始。

8.1.1 QueryResult

QueryResult，代表单 query 搜索，每个 QueryResult 中有 0 个或多个 Hit 对象。我们来看看 BLAST 文件是什么样的：

```
>>> 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...

```

虽然我们才接触对象模型的皮毛，但是你已经可以看到一些有用的信息了。通过调用 “QueryResult” 对象的 `print` 方法，你可以看到：

- 程序的名称和版本 (blastn version 2.2.27+)
- query 的 ID，描述和序列长度 (ID 是 42291，描述是 ‘mystery_seq’，长度是 61)
- 搜索的目标数据库 (refseq_rna)
- hit 结果的快速预览。对于我们的查询序列，有 100 个可能的 hit (表格中表示的是 0-99) 对于每个 hit，我们可以看到它包含的高分比对片段 (HSP)，ID 和一个片段描述。注意，`Bio.SearchIO` 截断了表格，只显示 0-29，然后是 97-99。

现在让我们用同样的步骤来检查 BLAT 的结果：

```

>>> 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>

```

马上可以看到一些不同点。有些是由于 BLAT 使用 PSL 格式储存它的信息，稍后会看到。其余是由于 BLAST 和 BLAT 搜索的程序和数据库之间明显的差异造成的：

- 程序名称和版本。`Bio.SearchIO` 知道程序是 BLAST，但是在输出文件中没有信息显示程序版本，所以默认是 ‘<unknown version>’。
- query 的 ID，描述和序列的长度。注意，这些细节和 BLAST 的细节只有细小的差别，ID 是 ‘mystery_seq’ 而不是 42991，缺少描述，但是序列长度仍是 61。这实际上是文件格式本身导致的差异。BLAST 有时创建自己的 query ID 并且用你的原始 ID 作为序列描述。
- 目标数据库是未知的，因为 BLAT 输出文件没提到相关信息。
- 最后，hit 列表完全不同。这里，我们的查询序列只命中到 ‘chr19’ 数据库条目，但是我们可以看到它含有 17 个 HSP 区域。这并不让人诧异，考虑到我们使用的是不同的程序，并且这些程序都有自己的数据库。

所有通过调用 `print` 方法看到的信息都可以单独地用 Python 的对象属性获得（又叫点标记法）。同样还可以用相同的方法获得其他格式特有的属性。

```
>>> 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
```

想获得一个可访问属性的完整列表，可以查询每个格式特有的文档。这些是 [BLAST](#) [BLAT](#)。

已经知道了在 `QueryResult` 对象上可以调用 `print` 方法，让我们研究的更深一些。`QueryResult` 到底是什么？就 Python 对象来说，`QueryResult` 混合了列表和字典的特性。换句话说，也就是一个包含了列表和字典功能的容器对象。

和列表以及字典一样，`QueryResult` 对象是可迭代的。每次迭代返回一个 `hit` 对象：

```
>>> 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)
...
```

要得到 `QueryResult` 对象有多少 `hit`，可以简单调用 Python 的 `len` 方法：

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

同列表类似，你可以用切片来获得 `QueryResult` 对象的 `hit`：

```
>>> 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)
```

要得到多个 `hit`，你同样可以对 `QueryResult` 对象作切片。这种情况下，返回一个包含被切 `hit` 的新 `QueryResult` 对象：

```
>>> 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 ...
```

同字典类似，可以通过 `ID` 获取 `hit`。如果你知道一个特定的 `hit ID` 存在于一个搜索结果中时，特别有用：

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

你可以用 `hits` 方法获得完整的 Hit 对象，也可以用 `hit.keys` 方法获得完整的 Hit ID：

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

如果你想确定一个特定的 hit 是否存在于查询结果中该怎么做呢？可以用 `in` 关键字作一个简单的成员检验：

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

有时候，只知道一个 hit 是否存在是不够的，你可能也会想知道 hit 的排名。 `index` 方法可以帮助你：

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

记住，我们用的是 Python 风格的索引，是从 0 开始。这代表 hit 的排名是 23 而不是 22。

同样，注意你看的 hit 排名是基于原始搜索输出文件的本来顺序。不同的搜索工具可能会基于不同的标准排列 hit。

如果原本的 hit 排序不合你意，可以用 `QueryResult` 对象的 `sort` 方法。它和 Python 的 `list.sort` 方法很相似，只是有个是否创建一个新的排序后的 `QueryResult` 对象的选项。

这里有个用 `QueryResult.sort` 方法对 hit 排序的例子，这个方法基于每个 hit 的完整序列长度。对于这个特殊的排序，我们设置 `in_place` 参数等于 `False`，这样排序方法会返回一个新的 `QueryResult` 对象，而原来的对象是未排序的。我们同样可以设置 `reverse` 参数等于 “`True`” 以递减排序。

```
>>> for hit in blast_qresult[:5]:      # id and sequence length of the first five hits
...     print 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 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
```

有 `in_place` 参数的好处是可以保留原本的顺序，后面会用到。注意这不是 `QueryResult.sort` 的默认行为，需要我们明确地设置 `in_place` 为 `True`。

现在，你已经知道使用 `QueryResult` 对象。但是，在我们学习 `Bio.SearchIO` 模块下个对象前，先了解下可以让 `QueryResult` 对象更易使用的两个方法：`filter` 和 `map` 方法。

如果你对 Python 的列表推导式、generator 表达式或内建的 `filter` 和 `map` 很熟悉，就知道（不知道就是看看吧！）它们在处理 list-like 的对象时有多有用。你可以用这些内建的方法来操作 `QueryResult` 对象，

但是这只对正常 list 有效，并且可操作性也会受到限制。

这就是为什么 QueryResult 对象提供自己特有的 filter 和 map 方法。对于 filter 有相似的 hit_filter 和 hsp_filter 方法，从名称就可以看出，这些方法过滤 QueryResult 对象的 Hit 对象或者 HSP 对象。同样的，对于 map，QueryResult 对象同样提供相似的 hit_map 和 hsp_map 方法。这些方法分别应用于 QueryResult 对象 hit 或者 HSP 对象。

让我们来看看这些方法的功能，从 hit_filter 开始。这个方法接受一个回调函数，这个函数检验给定的 Hit 是否符合你设定的条件。换句话说，这个方法必须接受一个单独 Hit 对象作为参数并且返回 True 或 False。

这里有个用 hit_filter 筛选出只有一个 HSP 的 Hit 对象的例子：

```
>>> 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 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 和 hit_filter 功能相同，只是它过滤每个 hit 中的 HSP 对象，而不是 Hit。

对于 map 方法，同样接受一个回调函数作为参数。但是回调函数返回修改过的 Hit 或 HSP 对象（取决于你是否使用 hit_map 或 hsp_map 方法），而不是返回 True 或 False。

来看一个用 hit_map 方法来重命名 hit ID 的例子：

```
>>> def map_func(hit):
...     hit.id = hit.id.split('|')[3]      # renames 'gi|301171322|ref|NR_035857.1|' to 'NR_035857.1'
...     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
```

同样的，hsp_map 和 hit_map 作用相似，但是作用于 HSP 对象而不是 Hit 对象。

8.1.2 Hit

Hit 对象代表从单个数据库获得所有查询结果。在 Bio.SearchIO 对象等级中是二级容器。它们被包含在 QueryResult 对象中，同时它们又包含 HSP 对象。

看看它们是什么样的，从我们的 BLAST 搜索开始：

```
>>> 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]
```

可以看到我们获得了必要的信息：

- query 的 ID 和描述信息。一个 hit 总是和一个 query 绑定，所以我们同样希望记录原始 query。这些值可以通过 query_id 和 query_description 属性从 hit 中获取。
- 我们同样得到了 hit 的 ID、描述和序列全长。它们可以分别通过 id, description, 和 seq.len 获取。
- 最后，有一个 hit 含有的 HSP 的简短信息表。在每行中，HSP 重要信息被列出来：HSP 索引，e 值，得分，长度（包括 gap），query 坐标和 hit 坐标。

现在，和 BLAT 结果作对比。记住，在 BLAT 搜索结果中，我们发现有一个含有 17HSP 的 hit。

```
>>> 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]
```

我们得到了和前面看到的 BLAST hit 详细程度相似的结果。但是有些不同点需要解释：

- e-value 和 bit score 列的值。因为 BLAT HSP 没有 e-values 和 bit scores，默认显示“?”。
- span 列是怎么回事呢？span 值本来是显示完整的比对长度，包含所有的残基和 gap。但是 PSL 格式目前还不支持这些信息并且 Bio.SearchIO 也不打算去猜它到底是多少，所有我们得到了和 e-value 以及 bit score 列相同的“?”。

就 Python 对象来说，Hit 和列表行为最相似，但是额外含有 HSP。如果你对列表熟悉，在使用 Hit 对象是不会遇到困难。

和列表一样，Hit 对象是可迭代的，并且每次迭代返回一个 HSP 对象：

```
>>> 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)
```

你可以对 Hit 对象调用 len 方法查看它含有多少个 HSP 对象：

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

你可以对 Hit 对象作切片取得单个或多个 HSP 对象，和 QueryResult 一样，如果切取多个 HSP，会返回包含被切片 HSP 的一个新 Hit 对象。

```
>>> 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]
```

你同样可以对一个 Hit 里的 HSP 排序，和你在 QueryResult 对象中看到的方法一样。

最后，同样可以对 Hit 对象使用 filter 和 map 方法。和 QueryResult 不同，Hit 对象只有一种 filter (Hit.filter) 和一种 map (Hit.map)。

8.1.3 HSP

HSP (高分片段) 代表 hit 序列中的一个区域，该区域包含对于查询序列有意义的比对。它包含了你的查询序列和一个数据库条目之间精确的匹配。由于匹配取决于序列搜索工具的算法，HSP 含有大部分统计信息，这些统计是由搜索工具计算得到的。这使得不同搜索工具的 HSP 对象之间的差异和你在 QueryResult 以及 Hit 对象看到的差异更加明显。

我们来看看 BLAST 和 BLAT 搜索的例子。先看 BLAST HSP：

```
>>> 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 - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG
      |||
Hit - CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG

```

和 `QueryResult` 以及 `Hit` 类似，调用 `HSP` 的 `print` 方法，显示细节：

- 有 `query` 和 `hit` 的 ID 以及描述。我们需要这些来识别我们的 `HSP`。
- 我们同样得到了 `query` 和 `hit` 序列的匹配范围。这里用的的切片标志着范围的表示是使用 Python 的索引风格（从 0 开始，半开区间）。圆括号里的数字表示正负链。这里，两条序列都是正链。
- 还有一些简短统计：`e-value` 和 `bitscore`。
- 还有一些 `HSP` 片段的信息。现在可以忽略，稍后会解释。
- 最后，还有 `query` 和 `hit` 的比对。

这些信息可以用点标记从它们本身获得，和 `Hit` 以及 `QueryResult` 相同：

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

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

它们并不是仅有的属性，`HSP` 对象有一系列的属性，使得获得它们的具体信息更加容易。下面是一些例子：

```

>>> 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

```

查看 [HSP 文档](#) 获取完整的属性列表。

不仅如此，每个搜索工具通常会对它的 `HSP` 对象作统计学或其他细节计算。例如，一个 XML BLAST 搜索同样输出 `gap` 以及相同的残基数量。这些属性可以像这样被获取：

```

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

```

这些细节是格式特异的；它们可能不会出现在其他的格式中。要知道哪些细节在给定的序列搜索工具中是存在的，你应该查看那种格式的在 `Bio.SearchIO` 中的文档。或者可以用 `...dict...keys()` 获得快速列表：

```

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

```

最后，你可能已经注意到了，我们 `HSP` 的 `query` 和 `hit` 属性不只是规律字符串：

```

>>> blast_hsp.query
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG', DNAAlphabet()), id='42291', name=

```

```
>>> blast_hsp.hit
SeqRecord(seq=Seq('CCCTCTACAGGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTCTTCCTTT...GGG', DNAAphabet()), id='gi|262205317|
```

它们是你已经在第4章看到过的 SeqRecord 对象！意味着你可以对 SeqRecord 对象做的各种有趣的事同样适用于 HSP.query 和 HSP.hit 对象。

现在 HSP 对象有个 alignment 属性（一个 MultipleSeqAlignment 对象）应该不会让你感到惊讶：

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

探索完 BLAST HSP 对象，让我们看看来自 BLAT 结果的不一样的 HSP。我们将对它调用 print 方法：

```
>>> 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: evalute ?; bitscore ?
Fragments: 1 (? columns)
```

一些输出你应该已经猜到了。我们得到了查询序列、hit ID、描述以及序列坐标。evalute 和 bitscore 的值是“?”，因为 BLAT HSP 并没有这些属性。但是最大的不同是你看不到任何的序列比对展示。如果你看的更仔细，PSL 格式本身并没有任何的 hit 和 query 序列，所以 Bio.SearchIO 不会创建任何序列或者比对对象。如果你尝试获取 HSP.query，HSP.hit，或者 HSP.aln 属性会怎么样呢？你会得到这些属性的默认值 None：

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

这并不影响其他的属性。例如，你仍然可以获取 query 和 hit 比对的长度。尽管不显示任何的属性，但是 PSL 格式还是有这些信息的，所以 Bio.SearchIO 可以抽提出这些信息。

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

其他格式特异的属性同样被展示出来：

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

到目前为止，一切还不错？当你看到 BLAT 结果中不同的 HSP 时，事情变得更有趣了。你可能会回想起在 BLAT 搜索中，有时我们把结果分成‘blocks’。这些区块是必需比对片段，可能会有些内含子在它们之间。

让我们看看 Bio.SearchIO 怎么处理包含多个区块的 BLAT HSP：

```
>>> 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]
```

怎么回事？我们仍然得到了一些必要的信息：ID，描述信息，坐标和快速统计，和你前面看到的一样。但是片段信息完全不同。我们得到了有两行数据的表格，而不是显示'Fragment: 1'。

这就是 Bio.SearchIO 处理含有多片段 HSP 的方式。和前面提到的一样，一个 HSP 比对可能会被内含子分成多个片段。内含子不是 query-hit 匹配的一部分，所以它们不能被当成 query 或 hit 序列的一部分。但是，它们确实影响我们处理序列坐标，所以我们不能忽视。

看看上面的 HSP 的 hit 坐标。在 Hit range 区域，我们看到坐标是 [54233104:54264463]。但是看看表格中的行，我们发现不是坐标跨度的所有区域都能匹配我们的 query。特殊的是，间断区域从 54233122 到 54264420。

你可能会问，为什么 query 坐标好像是邻近的？这是很好的。在这个例子中，query 是连续的（无间断区域），但是 hit 却不是。

所有的这些属性都是可以直接从 HSP 获取的，通过这样的方式：

```
>>> 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]
```

这些属性中大多数都不能简单地从 PSL 文件获得，但是当你分析 PSL 文件时，Bio.SearchIO 会动态地帮你计算。它需要的只是每个片段的开始和结束坐标。

query，hit，和 aln 属性又是什么情况？如果 HSP 含有多片段，你就不能使用这些属性，因为它们只取回单个 SeqRecord 或 MultipleSeqAlignment 对象。但是，你可以用相应的 *_all 方法：query_all，hit_all，和 aln_all。这些属性会返回包含每个 HSP 片段的 SeqRecord 或 MultipleSeqAlignment 对象的列表。还有其他相同功能的属性，也就是只对只有一个片段的 HSP 有效。查看 HSP 文档 获得完整的列表。

最后，想要检查是否是多片段 HSP，你可以用 is_fragmented 属性：

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

在进入下部分之前，你只需要了解我们可以对 HSP 对象使用切片，和 QueryResult 或 Hit 对象一样。当你使用切片的时候，会返回一个 HSPFragment 对象。

8.1.4 HSP 片段

HSPFragment 代表 query 和 hit 之间单个连续匹配。应该把它当作对象模型和搜索结果的核心，因为它决定你的搜索是否有结果。

在多数情况下，你不必直接处理 HSPFragment 对象，因为没有那么多搜索工具断裂它们的 HSP。当你确实需要处理它们时，需要记住的是 HSPFragment 对象要被写地尽量压缩。在多数情况下，它们仅仅包含直接与序列有关的属性：正负链，阅读框，字母表，位置坐标，序列本身以及它们的 ID 和描述。

当你调用 HSPFragment 对象的 print 方法时，这些属性可以非常简单地显示出来。这里有个从我们 BLAST 搜索得到的例子：

```
>>> 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 - CCCTCTACAGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG
      |||
Hit - CCCTCTACAGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTTAGAGGG
```

在这个水平上，BLAT 和 BLAST 片段看起来非常相似，除了没有出现的 query 和 hit 序列：

```
>>> 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)
```

在所有情况下，这些属性都可以通过我们最爱的点标记访问。一些例子：

```
>>> 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('CCCTCTACAGGAAGCGCTTTCTGTTGTCTGAAAGAAAAGAAAGTGCTTCCTTT...GGG', DNAAlphabet()), id='gi|262205317|
```

8.2 一个关于标准和惯例的注意事项

在我们进入到主要功能前，你需要知道 Bio.SearchIO 使用的一些标准。如果你已经接触过多序列搜索工具，你可能必须面对每个程序处理事情方式不同的问题，如序列位置坐标。这可能不是一个令人高兴的经历，因为这些搜索工具通常有它们自己的标准。例如，一种工具可能使用“从 1 开始”(one-based) 的坐标，而其他工具使用“从 0 开始”(zero-based) 的坐标。或者，一种程序在处理负链时，可能会反转开始和结束坐标，而其他程序确不会。简而言之，会产生一些必须要处理的混乱。

我们意识到这种问题，并且打算在 Bio.SearchIO 中解决。毕竟，Bio.SearchIO 的目标之一就是创建一个通用简单的接口来处理多种不同的搜索输出文件。意味着要制定一个超越你所见的对象模型的标准。

现在，你可能抱怨，“不要又来一个标准”。好吧，最后我们必须选择一个标准，这是必须的。并且，我们并不是创造一个全新的事物；只是采用一个我们觉得对 Python 使用者最好的标准（这是 Biopython，毕竟）。

在使用 Bio.SearchIO 时你可以认为有个三个隐含的标准：

- 第一个适用于序列坐标。在 Bio.SearchIO 模块中，所有序列坐标遵循 Python 的坐标风格：从 0 开始，半开区间。例如，在一个 BLAST XML 输出文件中，HSP 的起始和结束坐标是 10 和 28，它们在 Bio.SearchIO 中将变成 9 和 28。起始坐标变成 9 因为 Python 中索引是从 0 开始，而结束坐标仍然是 28 因为 Python 索引删除了区间中最后一个项目。
- 第二个是关于序列坐标顺序。在 Bio.SearchIO 中，开始坐标总是小于或等于结束坐标。但是这不是在所有的序列搜索工具中都始终适用。因为当序列为负链时，起始坐标会更大一些。
- 最后一个标准是关于链和阅读框的值。对于链值，只有四个可选值：1（正链），-1（负链），0（蛋白序列），和 None（无链）。对于阅读框，可选值是从 -3 至 3 的整型以及 None。

注意，这些标准只是存在于 Bio.SearchIO 对象中。如果你把 Bio.SearchIO 对象写入一种输出格式，Bio.SearchIO 会使用该格式的标准来输出。它并不强制它的标准到你的输出文件。

8.3 读取搜索输出文件

有两个方法，你可以用来读取搜索输出文件到 Bio.SearchIO 对象：read 和 parse。它们和其他亚模块如 Bio.SeqIO 或 Bio.AlignIO 中的 read 和 parse 方法在本质上是相似的。你都需要提供搜索输出文件名和文件格式名，都是 Python 字符串类型。你可以查阅文档来获得 Bio.SearchIO 可以识别的格式清单。

Bio.SearchIO.read 用于读取单 query 的搜索输出文件并且返回一个 QueryResult 对象。你在前面的例子中已经看到过 read 的使用了。你没看到的是，read 同样接受额外的关键字参数，取决于文件的格式。

这里有一些例子。在第一个例子中，我们和前面一样用 read 读 BLAST 表格输出文件。在第二个例子中，我们用一个关键字来修饰，所以它分析带有注释的 BLAST 表格变量。

```
>>> 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)
```

这些关键字在不同的文件格式中是不一样的。查看格式文档，看看它是否有关键字参数来控制它的分析器行为。

对于 Bio.SearchIO.parse，是用来读取含有任意数量 query 的搜索输出文件。这个方法返回一个 generator 对象，在每次迭代中 yield 一个 QueryResult 对象。和 Bio.SearchIO.read 一样，它同样接受格式特异的关键字参数：

```
>>> 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
```

8.4 用索引处理含有大量搜索输出的文件

有时，你得到了一个包含成百上千个 query 的搜索输出文件要分析，你当然可以使用 `Bio.SearchIO.parse` 来处理，但是如果你仅仅需要访问少数 query 的话，效率是及其低下的。这是因为 `parse` 会分析所有的 query，直到找到你感兴趣。

在这种情况下，理想的选择是用 `Bio.SearchIO.index` 或 `Bio.SearchIO.index.db` 来索引文件。如果名字听起来很熟悉，是因为你之前已经见过了，在章节 5.4.2。这些方法和 `Bio.SeqIO` 中相应的方法行为很相似，只是多了些格式特异的關鍵字参数。

这里有一些例子。你可以只用文件名和格式名来 `index`

```
>>> 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)
```

或者依旧使用格式特异的關鍵字参数：

```
>>> 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)
```

或者使用 `key_function` 参数，和 `Bio.SeqIO` 中一样：

```
>>> 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)
```

`Bio.SearchIO.index.db` 和 `index` 作用差不多，不同的只是它把 query 偏移量写入一个 SQLite 数据库文件中。

8.5 写入和转换搜索输出文件

有时候，读取一个搜索输出文件，作些调整并写到一个新的文件是很有用的。`Bio.SearchIO` 提供了一个 `write` 方法，让你可以准确地完成这种工作。它需要的参数是：一个可迭代返回 `QueryResult` 的对象，输出文件名，输出文件格式和一些可选的格式特异的關鍵字参数。它返回一个 4 项目的元组，分别代表被写入的 `QueryResult`，`Hit`，`HSP`，和 `HSPFragment` 对象的数量。

```
>>> 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)
```


你应该注意，不同的文件格式需要 `QueryResult`，`Hit`，`HSP` 和 `HSPFragment` 对象的不同属性。如果这些属性不存在，那么将不能写入。也就是，你想写入的格式可能有时也会失效。举个例子，如果你读取一个 BLASTXML 文件，你就不能将结果写入 PSL 文件，因为 PSL 文件需要一些属性，而这些属性 BLAST 却不能提供（如重复匹配的数量）。如果你确实想写到 PSL，可以手工设置这些属性。

和 `read`，`parse`，`index` 和 `index_db` 相似，`write` 同样接受格式特异的关键字参数。查阅文档获得 `Bio.SearchIO` 可写格式和这些格式的参数的完整清单。

最后，`Bio.SearchIO` 同样提供一个 `convert` 方法，可以理解为 `Bio.SearchIO.parse` 和 `Bio.SearchIO.write` 的简单替代方法。使用 `convert` 方法的例子如下：

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

因为 `convert` 使用 `write` 方法，所以只有所有需要的属性都存在时，格式转换才能正常工作。这里由于 BLAST XML 文件提供 BLAST 表格文件所需的所有默认值，格式转换才能正常完成。但是，其他格式转换就可能不会正常工作，因为你需要先手工指定所需的属性。

第 9 章访问 NCBI ENTREZ 数据库

Entrez (<http://www.ncbi.nlm.nih.gov/Entrez>) 是一个给客户提供 NCBI 各个数据库 (如 PubMed, GeneBank, GEO 等等) 访问的检索系统。用户可以通过浏览器手动输入查询条目访问 Entrez, 也可以使用 Biopython 的 Bio.Entrez 模块以编程方式访问来访问 Entrez。如果使用第二种方法, 用户用一个 Python 脚本就可以实现在 PubMed 里面搜索或者从 GenBank 下载数据。

Bio.Entrez 模块利用了 Entrez Programming Utilities (也称作 EUtils), 包含八个工具, 详情请见 NCBI 的网站: <http://www.ncbi.nlm.nih.gov/entrez/utils/>. 每个工具都能在 Python 的 Bio.Entrez 模块中找到对应函数, 后面会详细讲到。这个模块可以保证用来查询的 URL 的正确性, 并且向 NCBI 要求的一样, 每三秒钟查询的次数不超过一。

EUtils 返回的输出结果通常是 XML 格式, 我们有以下不同的方法来解析这种类型的输入文件:

1. 使用 Bio.Entrez 解析器将 XML 输出的解析成 Python 对象;
2. 使用 Python 标准库中的 DOM (Document Object Model) 解析器;
3. 使用 Python 标准库中的 SAX (Simple API for XML) 解析器;
4. 把 XML 输出当做原始的文本文件, 通过字符串查找和处理来进行解析;

对于 DOM 和 SAX 解析器, 可以查看 Python 的文档. Bio.Entrez 中使用到的解析器将会在下面讨论.

NCBI 使用 DTD (Document Type Definition) 文件来描述 XML 文件中所包含信息的结构. 大多数 NCBI 使用的 DTD 文件格式都包含在了 Biopython 发行包里。当 NCBI Entrez 读入一个 XML 格式的文件的时候, Bio.Entrez 将会使用 DTD 文件。

有时候, 你可能会发现与某种特殊的 XML 相关的 DTD 文件在 Biopython 发行包里面不存在。当 NCBI 升级它的 DTD 文件的时候, 这种情况可能发生。如果发生这种情况, Entrez.read 将会显示丢失的 DTD 文件名字和 URL 的警示信息。解析器会通过互联网获取缺失的 DTD 文件, 让 XML 的分析继续正常进行。如果本地存在对应的 DTD 文件的话, 处理起来会更快。因此, 为了更快的处理, 我们可以通过警示信息里面的 URL 来下载对应的 DTD 文件, 将文件放在 DTD 文件默认存放的文件夹 `...site-packages/Bio/Entrez/DTDs`。如果你没有权限进入这个文件夹, 你也可以把 DTD 文件放到 `~/biopython/Bio/Entrez/DTDs` 这个目录, `~` 表示的是你的 Home 目录。因为这个目录会先于 `...site-packages/Bio/Entrez/DTDs` 被解析器读取, 所以当 `...site-packages/Bio/Entrez/DTDs` 下面的 DTD 文件过时的时候, 你也可以将最新版本的 DTD 文件放到 Home 目录的那个文件夹下面。当然也有其他方案, 如果你是通过源码来安装的 Biopython, 你可以将 DTD 文件放到源码的 `Bio/Entrez/DTDs` 文件夹下, 然后重新安装 Biopython。这样会将新的 DTD 文件和之前的一样地安装到正确的位置。

Entrez Programming Utilities 也可以生成其他格式的输出文件, 比如 Fasta、序列数据库里面的 GenBank 文件格式或者文献数据库里面的 MedLine 格式, 更多内容将会在章节 9.12 中讨论。

9.1 Entrez 简介

在我们通过 Biopython 访问 NCBI 的线上资源（通过 Bio.Entrez 或者其他模块）的时候，请先阅读 NCBI 的 [Entrez 用户规范](#)。如果 NCBI 发现你在滥用他们的系统，他们会禁止你的访问。

详细规范如下：

- 对任何连续超过 100 次的访问请求，请在周末时间或者避开美国的使用高峰时间。这个取决于你是否遵从。
- 使用这个网址 <http://eutils.ncbi.nlm.nih.gov>，而不是通常的 NCBI 网址。Biopython 使用的是这个网址。
- 每秒钟不要超过三次请求（比 2009 年年初的每三秒钟最多一次请求要宽松）。这个由 Biopython 自动强制实行。
- 使用 email 参数，这样如果遇到什么问题，NCBI 可以通过邮件联系到你。你可以在每次请求 Entrez 的时候明确的设置这个参数（例如，在参数列表中包含 email="A.N.Other@example.com"），或者你也可以设置一个全局的 email 地址：

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

Bio.Entrez 将会在每次向 Entrez 请求的时候使用这个邮件地址。请千万不要胡乱的填写邮件地址，不填写都比这要好。邮件的参数从 2010 年 6 月 1 日将是强制的参数。在过度使用的情况下，NCBI 会在封锁用户访问 E-utilities 之前尝试通过用户提供的邮件地址联系。

- 如果你是在一个大的软件包里面使用 Biopython 的，请通过 tool 这个参数明确说明。你既可以在每次请求访问 Entrez 的时候通过参数明确地指明使用的工具（例如，在参数列表中包含 tool="MyLocalScript"），或者你也可以设置一个全局的 tool 名称：

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

默认的 tool 名称是 Biopython。

- 对于大规模的查询请求，NCBI 也推荐使用他们的会话历史特性（WebEnv 会话 cookie 字符串，见章节 9.15）。只是这个稍微有点复杂。

最后，根据你的使用情况选择不同的策略。如果你打算下载大量的数据，最好使用其他的方法。比如，你想得到所有人的基因的数据，那么考虑通过 FTP 得到每个染色体的 GenBank 文件，然后将这些文件导入到你自己的 BioSQL 数据库里面去。（请见章节 18.5）。

9.2 EInfo: 获取 Entrez 数据库的信息

EInfo 为每个 NCBI 的数据库提供了条目索引，最近更新的时间以及可用的链接。此外，你可以很容易的使用 EInfo 通过 Entrez 获取所有数据库名字列表：

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

变量 result 现在包含了 XML 格式的数据库列表：

```
>>> print result
<?xml version="1.0"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM/DTD eInfoResult, 11 May 2002//EN"
```

```
"http://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>
```

因为这是一个相当简单的 XML 文件，我们可以简单的通过字符串查找提取里面所包含的信息。使用 Bio.Entrez 的解析器，我们可以直接将这个 XML 读入到一个 Python 对象里面去：

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

现在 record 是拥有一个确定键值的字典：

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

这个键对应的值存储了上面 XML 文件里面包含的数据库名字的列表：

```
>>> 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']
```

对于这些数据库，我们可以使用 EInfo 获得更多的信息：

```
>>> handle = Entrez.einfo(db="pubmed")  
>>> record = Entrez.read(handle)  
>>> record["DbInfo"]["Description"]  
'PubMed bibliographic record'  
>>> record["DbInfo"]["Count"]  
'17989604'  
>>> record["DbInfo"]["LastUpdate"]  
'2008/05/24 06:45'
```

通过 `record["DbInfo"].keys()` 可以获取存储在这个记录里面的其他信息。这里面最有用的信息之一是一个 ESearch 可用的搜索值列表：

```
>>> 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  
...
```

这是一个很长的列表，但是间接的告诉你在使用 PubMed 的时候，你可以通过 `Jones[AUTH]` 搜索作者，或者通过 `Sanger[AFFL]` 将作者范围限制在 Sanger Centre。这个会非常方便，特别是在你对某个数据库不太熟悉的时候。

9.3 ESearch: 搜索 Entrez 数据库

我们可以使用 `Bio.Entrez.esearch()` 来搜索任意的数据库。例如，我们在 PubMed 中搜索跟 Biopython 相关的文献：

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

在输出的结果中，我们可以看到七个 PubMed IDs（包括 19304878，这个是 Biopython 应用笔记的 PMID），你可以通过 EFetch 来获取这些文献（请见章节 9.6）。

你也可以通过 ESearch 来搜索 GenBank。我们将以在 `*Cypripedioideae*` orchids 中搜索 `*matK*` 基因为例，快速展示一下（请见章节 9.2 关于 EInfo：一种查明你可以在哪个 Entrez 数据库中搜索的方法）。

```
>>> handle = Entrez.esearch(db="nucleotide", term="Cypridiedioideae[Orgn] AND matK[Gene]")
>>> record = Entrez.read(handle)
>>> record["Count"]
'25'
>>> record["IdList"]
['126789333', '37222967', '37222966', '37222965', ..., '61585492']
```

每个 IDs(126789333, 37222967, 37222966, ...) 是 GenBank 的一个标识。请见章节 9.6 此章包含了怎样下载这些 GenBank 的记录的信息。

注意，不是像 Cypridiedioideae[Orgn] 这样在搜索的时候加上特定的物种名字，而是需要在搜索的时候使用 NCBI 的 taxon ID，像 txid158330[Orgn] 这样。这个并没有记录在 ESearch 的帮助页面上，NCBI 通过邮件回复解释了这个问题。你可以通过经常和 Entrez 的网站接口互动，来推断搜索条目的格式。例如，在基因组搜索的时候加上 complete[prop] 可以把结果限制在完成的基因组上。

作为最后一个例子，让我们获取一个 computational journal 名字列表：

```
>>> handle = Entrez.esearch(db="journals", term="computational")
>>> record = Entrez.read(handle)
>>> record["Count"]
'16'
>>> record["IdList"]
['30367', '33843', '33823', '32989', '33190', '33009', '31986',
 '34502', '8799', '22857', '32675', '20258', '33859', '32534',
 '32357', '32249']
```

同样，我们可以通过 EFetch 来获得关于每个 journal IDs 更多的消息。

ESearch 有很多有用的参数——参见 ESearch 帮助页面 来获取更多信息。

9.4 EPost: 上传 identifiers 的列表

EPost 上传在后续搜索中将会用到的 IDs 的列表，参见 EPost 帮助页面 来获取更多信息。通过 Bio.Entrez.epost() 函数可以在 Biopython 中实现。

为了举一个关于此用法的例子，假设你有一个想通过 EFetch 下载的 IDs 的长长的列表（可能是序列，也有可能是引用的其他内容）。当你通过 EFetch 发出下载请求的时候，你的 IDs 列表、数据库等，将会被转变成一个长的 URL，然后被发送到服务器。如果 IDs 列表很长，URL 也会很长，长的 URL 可能会断掉（比如，一些代理不能复制全部的内容）。

另外，你也可以把以上分成两步来完成，首先用 EPost 来上传 IDs 的列表（这个使用了一个内部的“HTML post”，而不是“HTML get”，避开了 long URL 可能产生的问题）。由于历史记录的支持，你可以使用 EFetch 来指向这个长的 IDs 列表，并且下载相关的数据。

让我们通过下面一个简单的例子来看看 EPost 是如何工作的——上传了一些 PubMed 的 IDs：

```
>>> 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"
"http://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>
```

返回的 XML 包含了两个重要的字符串，QueryKey 和 WebEnv，两个字符串一起确定了之前的历史记录。你可以使用其他的 Entrez 工具，例如 EFetch，来提取这些值：

```
>>> 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"]
```

第9.15 章节讲述了如何使用历史的特性。

9.5 ESummary: 通过主要的 IDs 来获取摘要

ESummary 可以通过一个 primary IDs 来获取文章的摘要（参见 [ESummary 帮助页面](#) 来获取更多信息）。在 Biopython 中，ESummary 以 Bio.Entrez.esummary() 的形式出现。根据上面的搜索结果，我们可以获得 ID 为 30367 杂志相关的更多信息：

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.esummary(db="journals", id="30367")
>>> record = Entrez.read(handle)
>>> record[0]["Id"]
'30367'
>>> record[0]["Title"]
'Computational biology and chemistry'
>>> record[0]["Publisher"]
'Pergamon,'
```

9.6 EFetch: 从 Entrez 下载更多的记录

当你想要从 Entrez 中提取完整的记录的时候，你可以使用 EFetch。在 [EFetch 的帮助页面](#) 可以查到 EFetch 可以起作用的数据库。

NCBI 大部分的数据库都支持多种不同的文件格式。当使用 Bio.Entrez.efetch() 从 Entrez 下载特定的某种格式的时候，需要 rettype 和或者 retmode 这些可选的参数。对于不同数据库类型不同的搭配在下面的网页中有描述：[NCBI efetch webpage](#) (例如：[literature](#), [sequences](#) and [taxonomy](#))。

一种常用的用法是下载 FASTA 或者 GenBank/GenPept 的文本格式 (接着可以使用 Bio.SeqIO 来解析, 参见 [5.3.1](#) 和 [9.6](#))。从上面 *Cypripedioideae* 的例子, 我们可以通过 Bio.Entrez.efetch 从 GenBank 下载记录 186972394。

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="gb", retmode="text")
>>> print handle.read()
LOCUS      EU490707                      1302 bp    DNA        linear    PLN 05-MAY-2008
DEFINITION Selenipedium aequinoctiale maturase K (matK) gene, partial cds;
            chloroplast.
ACCESSION  EU490707
VERSION    EU490707.1  GI:186972394
KEYWORDS   .
SOURCE     chloroplast Selenipedium aequinoctiale
ORGANISM   Selenipedium aequinoctiale
```



```

Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
Spermatophyta; Magnoliophyta; 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,C.L., Williams,N.H. and Moore,M.J.
TITLE Phylogenetic utility of ycf1 in orchids
JOURNAL Unpublished
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"
                        /db_xref="GI:186972395"
                        /translation="IFYEPVEIFGYDNKSSLVLVKRLITRMYQQNFLISSVNSNQKG
FWGHKHFSSSHFSQMVSEGFVILEIPFSSQLVSSLEKKIPKYQNLRSIHSIFPFL
EDKFLHLNYSVDLLIPHPHLEILVQILQCRIKDVPSLHLLRLLFHEYHNLNSLITSK
KFIYAFSKRKKRFLWLLYNSYVVECEYLFQFLRKQSSYLSTSSGVFLERTHLYVKIE
HLLVCCNSFQRILCFLKDPFMHYVRYQGKAILASKGTLILMKKWFHLVNFQSYFH
FWSQPYRIHIKQLSNYSFSFLGYFSSVLENHLVVRNQMLENSFIINLLTKKFDTIAPV
ISLIGLSKAQFCTVLGHPISKPIWTDSDSDILDRFCRICRNLCRYHSGSKKQVLY
RIKYILRLSCARTLARKHKSTVRTFMRLGSGLLEEFFMEEE"
ORIGIN
1 attttttacg aacctgtgga aatttttggT tatgacaata aatctagttt agtacttgtg
61 aaacgtttaa ttactcgaat gtatcaacag aattttttga tttcttcggt taatgattct
121 aacaaaaaag gattttgggg gcacaagcat tttttttcct 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 atgtaaaaaat agaacatctt
661 ctagtagtgt gttgtaattc ttttcagagg atcctatgct ttctcaagga tcctttcatg
721 cattatgttc gatatcaagg aaaagcaatt ctggcctcaa aggggaactct tattctgatg
781 aagaaatgga aatttcatct tgtgaatttt tggcaattct atttccactt ttggtctcaa
841 ccgtatagga ttcatataaa gcaattatcc aactattcct tctcttttct ggggtatttt
901 tcaagtgtac tagaaaaatca 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
//
```

参数 `rettype="gb"` 和 `retmode="text"` 让我们下载的数据为 GenBank 格式。

需要注意的是直到 2009 年，Entrez EFetch API 要求使用“genbank”作为返回类型，然而现在 NCBI 坚持使用官方的“gb”或“gbwithparts”（或者针对蛋白的“gp”）返回类型。同样需要注意的是，直到 2012 年 2 月，Entrez EFetch API 默认返回格式为纯文本格式文件，现在默认为 XML 格式。

作为另外的选择，你也可以使用 `rettype="fasta"` 来获取 Fasta 格式的文件；参见 [EFetch Sequences 帮助页面](#)。记住，可选的数据格式决定于你要下载的数据库——请参见 [EFetch 帮助页面](#)。

如果你要获取记录的格式是 Bio.SeqIO 所接受的一种格式（见第 5 章），你可以直接将其解析为一个 SeqRecord：

```
>>> from Bio import Entrez, SeqIO
>>> handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="gb", retmode="text")
>>> record = SeqIO.read(handle, "genbank")
>>> handle.close()
>>> print record
ID: EU490707.1
Name: EU490707
Description: Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast.
Number of features: 3
...
Seq('ATTTTTCGAACTGTGGAAATTTTGGTTATGACAATAAATCTAGTTTAGTA...GAA', IUPACAmbiguousDNA())
```

需要注意的是，一种更加典型的用法是先把序列数据保存到一个本地文件，然后使用 Bio.SeqIO 来解析。这样就避免了在运行脚本的时候需要重复的下载同样的文件，并减轻 NCBI 服务器的负载。例如：

```
import os
from Bio import SeqIO
from Bio import Entrez
Entrez.email = "A.N.Other@example.com" # Always tell NCBI who you are
filename = "gi_186972394.gb"
if not os.path.isfile(filename):
    # Downloading...
    net_handle = Entrez.efetch(db="nucleotide", id="186972394", rettype="gb", retmode="text")
    out_handle = open(filename, "w")
    out_handle.write(net_handle.read())
    out_handle.close()
    net_handle.close()
    print "Saved"

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

为了得到 XML 格式的输出，你可以使用 Bio.Entrez.read() 函数和参数 `retmode="xml"` 进行解析，：

```
>>> from Bio import Entrez
>>> handle = Entrez.efetch(db="nucleotide", id="186972394", retmode="xml")
>>> record = Entrez.read(handle)
>>> handle.close()
>>> record[0]["GBSeq_definition"]
'Selenipedium aequinoctiale maturase K (matK) gene, partial cds; chloroplast'
```

```
>>> record[0]["GBSeq_source"]
'chloroplast Selenipedium aequinoctiale'
```

就像这样处理数据。例如解析其他数据库特异的文件格式（例如，PubMed 中用到的 MEDLINE 格式），请参见章节 9.12。

如果你想使用 Bio.Entrez.esearch() 进行搜索，然后用 Bio.Entrez.efetch() 下载数据，那么你需要用到 WebEnv 的历史特性，请参见章节 9.15。

9.7 ELink: 在 NCBI Entrez 中搜索相关的条目

ELink，在 Biopython 中是 Bio.Entrez.elink()，可以用来在 NCBI Entrez 数据库中寻找相关的条目。例如，你可以使用它在 gene 数据库中寻找核苷酸条目，或者其他很酷的事情。

让我们使用 ELink 来在 2009 年的 *Bioinformatics* 杂志中寻找与 Biopython 应用相关的文章。这篇文章的 PubMed ID 是 19304878：

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"
>>> pmid = "19304878"
>>> record = Entrez.read(Entrez.elink(dbfrom="pubmed", id=pmid))
```

变量 record 包含了一个 Python 列表，列出了已经搜索过的数据库。因为我们特指了一个 PubMed ID 来搜索，所以 record 只包含了一个条目。这个条目是一个字典变量，包含了我们需要寻找的条目的信息，以及能搜索到的所有相关的内容：

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

键 "LinkSetDb" 包含了搜索结果，将每个目标数据库保存为一个列表。在我们这个搜索中，我们只在 PubMed 数据库中找到了结果（尽管已经被分到了不同的分类）：

```
>>> len(record[0]["LinkSetDb"])
5
>>> for linksetdb in record[0]["LinkSetDb"]:
...     print linksetdb["DbTo"], linksetdb["LinkName"], len(linksetdb["Link"])
...
pubmed pubmed_pubmed 110
pubmed pubmed_pubmed_combined 6
pubmed pubmed_pubmed_five 6
pubmed pubmed_pubmed_reviews 5
pubmed pubmed_pubmed_reviews_five 5
```

实际的搜索结果被保存在键值为 "Link" 的字典下。在标准搜索下，总共找到了 110 个条目。让我们现在看看我们第一个搜索结果：

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

这个就是我们搜索的文章，从中并不能看到更多的结果，所以让我们来看看我们的第二个搜索结果：

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

这个 PubMed ID 为 14530660 的文章是关于 Biopython PDB 解析器的。

我们通过一个循环来打印出所有的 PubMed IDs :

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

现在漂亮极了,但是对我个人而言,我对某篇文章是否被引用过更感兴趣。好吧,ELink 也可以完成这个——至少对 PubMed Central 的杂志来说是这样的(请见章节9.15.3)。

关于 ELink 的帮助,请见 [ELink 帮助页面](#)。这是一个关于 link names 的整个的子页面,描述了不同的数据库可以怎样交叉的索引。

9.8 EGQuery: 全局搜索 - 统计搜索的条目

EGQuery 提供搜索字段在每个 Entrez 数据库中的数目。当我们只需要知道在每个数据库中能找到的条目的个数,而不需要知道具体搜索结果的时候,这个非常的有用(请见例子9.14.2)。

在这个例子中,我们使用 Bio.Entrez.egquery() 来获取跟“Biopython”相关的数目:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.egquery(term="biopython")
>>> record = Entrez.read(handle)
>>> for row in record["eGQueryResult"]: print row["DbName"], row["Count"]
...
pubmed 6
pmc 62
journals 0
...
```

请见 [EGQuery 帮助页面](#) 获得更多信息。

9.9 ESpell: 获得拼写建议

ESpell 可以检索拼写建议。在这个例子中,我们使用 Bio.Entrez.espell() 来获得 Biopython 正确的拼写:

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

请见 [ESpell 帮助页面](#) 获得更多信息。这个的主要用法是在使用 GUI 工具的时候为搜索的条目自动的提供拼写建议。

9.10 解析大的 Entrez XML 文件

Entrez.read 函数将 Entrez 返回的结果读取到一个 Python 对象里面去，这个对象被保存在内存中。对于解析太大的 XML 文件而内存不够时，可以使用 Entrez.parse 这个函数。这是一个生成器函数，它将一个一个的读取 XML 文件里面的内容。只有 XML 文件是一个列表对象的时候，这个函数才有用（换句话说，如果在一个内存无限的计算机上 Entrez.read 将返回一个 Python 列表）。

例如，你可以通过 NCBI 的 FTP 站点从 Entrez Gene 数据库中下载某个物种全部的条目作为一个文件。这个文件可能很大。作为一个例子，在 2009 年 9 月 4 日，文件 Homo_sapiens.ags.gz 包含了 Entrez Gene 数据库中人的序列，文件大小有 116576kB。这个文件是 ASN 格式，可以通过 NCBI 的 gene2xml 程序转成 XML 格式（请到 NCBI 的 FTP 站点获取更多的信息）：

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

XML 结果文件有 6.1GB. 在大多数电脑上尝试 Entrez.read 都会导致 MemoryError。

XML 文件 Homo_sapiens.xml 包含了一个 Entrez gene 记录的列表，每个对应于人的一个 Entrez 基因信息。Entrez.parse 将一个一个的读取这些记录。这样你可以通过遍历每个记录的方式打印或者存储每个记录相关的信息。例如，下面这个脚本遍历了 Entrez 基因里面的记录，打印了每个基因的数目和名字：

```
>>> from Bio import Entrez
>>> handle = open("Homo_sapiens.xml")
>>> records = Entrez.parse(handle)

>>> 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
```

将会打印以下内容：

```
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
...
```

9.11 错误处理

当解析 XML 文件的时候，可能出现一下三个错误：

- 这个文件可能不是以常规的 XML 文件格式开头；
- 这个文件可能不完整或者包含一些非 XML 格式的内容；

- 这个文件是正常的 XML 文件，但是包含和相关 DTD 文件无关的条目。

第一种情况会在，例如，你尝试把一个 Fasta 文件当做 XML 文件来处理时发生：

```
>>> from Bio import Entrez
>>> handle = open("NC_005816.fna") # a Fasta file
>>> record = Entrez.read(handle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/__init__.py", line 257, in read
    record = handler.read(handle)
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/Parser.py", line 164, in read
    raise NotXMLError(e)
Bio.Entrez.Parser.NotXMLError: Failed to parse the XML data (syntax error: line 1, column 0). Please make sure that
```

这时候，解析器找不到 `<?xml ...` 标签，而这是一个 XML 文件开始的标志，那么可以确定这个文件不是 XML 文件。

当你的文件是 XML 格式，但是是不完整的（例如，提前结束了），那么解析器会报 `CorruptedXMLError` 错误。下面这个是一个 XML 文件提前结束的例子：

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

这个会生成以下的日志文件：

```
>>> Entrez.read(handle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/__init__.py", line 257, in read
    record = handler.read(handle)
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/Parser.py", line 160, in read
    raise CorruptedXMLError(e)
Bio.Entrez.Parser.CorruptedXMLError: Failed to parse the XML data (no element found: line 16, column 0). Please mak

>>>
```

注意，报错信息告诉你在 XML 文件的什么位置检测到了错误。

如果 XML 文件当中包含有对应 DTD 文件中没有描述的标签的时候，会发生第三类错误。以下是这样一个 XML 文件的例子：

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

```

在这个文件里面，因为一些原因，`<DocsumList>`（还有一些其他的）标签没有在 DTD 文件 `eInfo_020511.dtd` 中列出来，XML 文件对应 DTD 文件的第二行会特别的描述出来。默认情况下，如果没有找到 DTD 文件中的标签，解析器会中止并报 `ValidationError` 错误。

```

>>> from Bio import Entrez
>>> handle = open("einfo3.xml")
>>> record = Entrez.read(handle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/__init__.py", line 257, in read
    record = handler.read(handle)
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/Parser.py", line 154, in read
    self.parser.ParseFile(handle)
  File "/usr/local/lib/python2.7/site-packages/Bio/Entrez/Parser.py", line 246, in startElementHandler
    raise ValidationError(name)
Bio.Entrez.Parser.ValidationError: Failed to find tag 'DocsumList' in the DTD. To skip all tags that are not repres

```

可选地，你可以让解析器跳过这样的标签，而不是报 `ValidationError` 错误。通过调用 `Entrez.read` 或者 `Entrez.parse` 并使参数 `validate` 等于 `False` 可以实现这个功能：

```

>>> from Bio import Entrez
>>> handle = open("einfo3.xml")
>>> record = Entrez.read(handle, validate=False)
>>>

```

当然，XML 文件中的 tag 没有出现在对应 DTD 文件中的信息，将不会在 `Entrez.read` 的返回记录中出现。

9.12 专用的解析器

函数 `Bio.Entrez.read()` 可以处理大部分（如果不是所有的话）Entrez 返回的 XML 文件。Entrez 也可以允许你通过其他格式来获取数据，有时候，这种方式在可读性上比 XML 文件格式更具优势（或者下载文件的大小）。

为了使用 `Bio.Entrez.efetch()` 函数从 Entrez 中提取一种特有的文件格式，需要指明 `rettype` 和或者 `retmode` 等可选参数。不同的组合在 [NCBI efetch 的页面](#)。有对不同数据库的描述。

一个显然的例子是，你可能更想以 FASTA 或者 GenBank/GenPept（这些可以通过 Bio.SeqIO 来处理，请见 5.3.1 和 9.6）纯文本形式下载序列。对于文献数据库，Biopython 包含了一个处理 PubMed 中使用的 MEDLINE 格式的解析器。

9.12.1 解析 Medline 记录

你可以在 Bio.Medline 中找到 Medline 的解析器。假设你想处理包含一个 Medline 记录的 pubmed_result1.txt 文件。你可以在 Biopython 的 Tests\Medline 目录下找到这个文件，这个文件内容如下所示：

```
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
...
```

我们首先打开文件，然后解析它：

```
>>> from Bio import Medline
>>> input = open("pubmed_result1.txt")
>>> record = Medline.read(input)
```

现在 record 将 Medline 记录以 Python 字典的形式保存起来：

```
>>> 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 customised 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.'
```

用于 Medline 记录的键值可以相当模糊，使用

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

可以做一个简单的总结。

为了解析包含多个 Medline 记录的文件，你可以使用 parse 函数来代替：

```
>>> from Bio import Medline
>>> input = open("pubmed_result2.txt")
>>> records = Medline.parse(input)
>>> for record in records:
```

```
...     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.

你可以通过 `Bio.Entrez.efetch` 来下载 Medline 记录，而不是保存在某个文件里。例如，让我们来查看 PubMed 里面跟 Biopython 相关的所有所有 Medline 记录：

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

现在我们使用 `Bio.Entrez.efetch` 来下载这些 Medline 记录：

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

这里，我们使 `rettype="medline"`，`retmode="text"` 来以纯文本形式的 Medline 格式来得到这些记录。现在我们使用 `Bio.Medline` 来解析这些记录：

```
>>> from Bio import Medline
>>> records = Medline.parse(handle)
>>> 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']
```

为了比对，我们展示了一个 XML 格式的例子：

```
>>> idlist = record["IdList"]
>>> handle = Entrez.efetch(db="pubmed",id=idlist,rettype="medline",retmode="xml")
>>> records = Entrez.read(handle)
>>> for record in records:
...     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.
```

需要注意的是，在上面这两个例子当中，为了简便我们混合使用了 `ESearch` 和 `EFetch`。在这种情形下，NCBI 希望你使用他们的历史记录特性，在下面章节中会讲到 [Section 9.15](#)。

9.12.2 解析 GEO 记录

GEO (Gene Expression Omnibus) 是高通量基因表达和杂交芯片数据的数据库。Bio.Geo 模块可以用来解析 GEO 格式的数据。

下面的代码展示了怎样将一个名称为 GSE16.txt 的 GEO 文件存进一个记录，并打印该记录：

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

你可以使用 ESearch 来搜索“gds”数据库 (GEO 数据集)：

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com" # Always tell NCBI who you are
>>> handle = Entrez.esearch(db="gds",term="GSE16")
>>> record = Entrez.read(handle)
>>> record["Count"]
2
>>> record["IdList"]
['200000016', '100000028']
```

通过 Entrez 网站，UID “200000016”是 GDS16，其他的 hit “100000028”是相关的平台。不幸的是，在写这份指南的时候，NCBI 貌似还不支持通过 Entrez 下载 GEO 文件（不论 XML 文件，还是 SOFT 格式的文件）。

然而，可以相当直接的通过 FTP <ftp://ftp.ncbi.nih.gov/pub/geo/> 来下载 GEO 文件。在这个例子当中，你需要的文件应该是 ftp://ftp.ncbi.nih.gov/pub/geo/DATA/SOFT/by_series/GSE16/GSE16_family.soft.gz（一个压缩文件，参见 Python 的 gzip 模块）。

9.12.3 解析 UniGene 记录

UniGene 是 NCBI 的转录组数据库，每个 UniGene 记录展示了该转录本在某个特定物种中相关的基因。一个典型的 UniGene 记录如下所示：

```
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 tumor| adult
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=44157214
...
SEQUENCE    ACC=AU099534.1; NID=g13550663; CLONE=HSI08034; END=5'; LID=8800; SEQTYPE=EST
//

```

这个记录展示了这个转录本 (如 SEQUENCE 行展示) 是来自人的 NAT2 基因, 编码 en N-acetyltransferase。PROTSIM 显示的是和 NAT2 显著相似的蛋白质, STS 展示的是基因组当中的 STS 位点。

我们使用 Bio.UniGene 模块来解析 UniGene 文件:

```

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

```

UniGene.read 返回的是一个包含一些和 UniGene 记录的字段相对应属性的 Python 对象。例如,

```

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

```

EXPRESS 和 RESTR_EXPR 两行被存储为字符串的 Python 列表:

```
['bone', 'connective tissue', 'intestine', 'liver', 'liver tumor', 'normal', 'soft tissue/muscle tissue tumor', 'adipose tissue']
```

跟 STS, PROTSIM, 和 SEQUENCE 相关的特有的对象被保存在如下键所对应的字典中:

```

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

```

和 PROTSIM、SEQUENCE 这两行相似。

我们使用 Bio.UniGene 中的 parse 函数来处理一个文件中包含多个 UniGene 记录的情况:

```

>>> from Bio import UniGene
>>> input = open("unigenerecords.data")
>>> records = UniGene.parse(input)
>>> for record in records:
...     print record.ID

```

9.13 使用代理

通常状况下, 你不需要使用代理, 但是如果你的网络有问题的时候, 我们有以下应对方法。在内部, Bio.Entrez 使用一个标准的 Python 库 urllib 来访问 NCBI 的服务器。这个将检查叫做 http_proxy 的环境变量来自动配置简单的代理服务。不幸的是, 这个模块不支持需要认证的代理。

你可以选择设定环境变量 http_proxy。同样, 你可以在 Python 脚本开头的地方设置这个参数, 例如:

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

参见 `urllib` 文档 获得更多信息。

9.14 实例

9.14.1 PubMed 和 Medline

如果你是在医药领域或者对人类的问题感兴趣（或者尽管并不感兴趣，大多数情况下也适用!），PubMed(<http://www.ncbi.nlm.nih.gov/PubMed/>) 是一个包含了各方面的非常优秀的资源。像其他的一样，我们希望能够通过 Python 脚本从中抓取一些信息。

在这个例子当中，我们要查询 PubMed 当中所有跟 Orchids 相关的文章（见 2.3 我们的动机）。我们首先看看有多少这样的文章：

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

现在我们使用 `Bio.Entrez.efetch` 这个函数来下载这 463 篇文章的 PubMed IDs：

```
>>> handle = Entrez.esearch(db="pubmed", term="orchid", retmax=463)
>>> record = Entrez.read(handle)
>>> idlist = record["IdList"]
>>> print idlist
```

返回值是一个 Python 列表，包含了所有和 orchids 相关文章的 PubMed IDs：

```
['18680603', '18665331', '18661158', '18627489', '18627452', '18612381',
'18594007', '18591784', '18589523', '18579475', '18575811', '18575690',
...]
```

这样我们就得到了这些信息，显然我们想要得到对应的 Medline records 和更多额外的信息。这里，我们将以纯文本的形式下载和 Medline records 相关的信息，然后使用 `Bio.Medline` 模块来解析他们：

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

注意 - 我们完成了一次搜索和获取，NCBI 更希望你在这种情况下使用他们的历史记录支持。请见章节 9.15。

请记住 `records` 是一个迭代器，所以你只能访问这些 `records` 一次。如果你想保存这些 `records`，你需要把他们转成列表：

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

现在让我们迭代这些 `records`，然后分别打印每一个 `record` 的信息：

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

这个的输出结果是这样的:

```
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
```

特别有意思的是作者的列表, 作者的列表会作为一个标准的 Python 列表返回。这使得用标准的 Python 工具操作和搜索变得简单。例如, 我们可以像下面的代码这样循环读取所有条目来查找某个特定的作者:

```
>>> 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"])
```

希望这个章节可以让你知道 Entrez 和 Medline 借口的能力和便利性和怎样同时使用他们。

9.14.2 搜索, 下载, 和解析 Entrez 核酸记录

这里我们将展示一个关于远程 Entrez 查询的简单例子。在 2.3 节, 我们讲到了使用 NCBI 的 Entrez 网站来搜索 NCBI 的核酸数据库来获得关于 *Cypripedioideae* 的信息。现在我们看看如何使用 Python 脚本自动的处理。在这个例子当中, 我们仅仅展示如何使用 Entrez 模块来连接, 获取结果, 解析他们。

首先, 我们在下载这些结果之前, 使用 EGQuery 来计算结果的数目。EGQuery 将会告诉我们在每个数据库中分别有多少搜索结果, 但在我们这个例子当中, 我们只对核苷酸感兴趣:

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.egetquery(term="Cypripedioideae")
>>> record = Entrez.read(handle)
>>> for row in record["egetqueryResult"]:
...     if row["DbName"]=="nuccore":
...         print row["Count"]
814
```

所以, 我们预期能找到 814 个 Entrez 核酸记录 (这是我在 2008 年得到的结果; 在未来这个结果应该会增加)。如果你得到了高的不可思议的结果数目时, 你可能得重新考虑是否需要下载所有的这些结果, 下载是我们的下一步:

```
>>> from Bio import Entrez
>>> handle = Entrez.esearch(db="nucleotide", term="Cypripedioideae", retmax=814)
>>> record = Entrez.read(handle)
```

在这里, record 是一个包含了搜索结果和一些辅助信息的 Python 字典。仅仅作为参考信息, 让我们看看在这些字典当中究竟存储了些什么内容:

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

首先, 让我们检查看看我们得到了多少个结果:

```
>>> print record["Count"]
'814'
```

这个结果是我们所期望的。这 814 个结果被存在了 record['IdList'] 中:

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

让我们看看前五个结果:

```
>>> print record["IdList"][:5]
['187237168', '187372713', '187372690', '187372688', '187372686']
```

我们可以使用 efetch 来下载这些结果。尽管你可以一个一个的下载这些记录, 但为了减少 NCBI 服务器的负载, 最好呢还是一次性的下载所有的结果。然而在这个情况下, 你应该完美的使用在后面章节 9.15 中会要讲到的历史记录特性。

```
>>> idlist = ",".join(record["IdList"][:5])
>>> print idlist
187237168,187372713,187372690,187372688,187372686
>>> handle = Entrez.efetch(db="nucleotide", id=idlist, retmode="xml")
>>> records = Entrez.read(handle)
>>> print len(records)
5
```

每个这样的 records 对应一个 GenBank record.

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

你可以用这个来快速的开始搜索——但是对于频繁的使用请见 9.15.

9.14.3 搜索、下载和解析 GenBank record

GenBank record 格式是保存序列信息、序列特征和其他相关信息非常普遍的一种方法。这种格式是从 NCBI 数据库 <http://www.ncbi.nlm.nih.gov/> 获取信息非常好的一种方式。

在这个例子当中，我们将展示怎样去查询 NCBI 数据库，根据 query 提取记录，然后使用 Bio.SeqIO 解析他们——在 5.3.1 中提到过这些。简单起见，这个例子 * 不会 * 使用 WebEnv 历史记录特性——请到 9.15 查看。

首先，我们想要查询找出要获取的记录的 ID。这里我们快速的检索我们最喜欢的一个物种 *Opuntia* (多刺的梨型仙人掌)。我们可以做一个快速的检索来获得所有满足要求的 GIs (GenBank 标志符)。首先我们看看有多少个记录：

```
>>> from Bio import Entrez
>>> Entrez.email = "A.N.Other@example.com"      # Always tell NCBI who you are
>>> handle = Entrez.egetquery(term="Opuntia AND rpl16")
>>> record = Entrez.read(handle)
>>> for row in record["eGQueryResult"]:
...     if row["DbName"]=="nuccore":
...         print row["Count"]
...
9
```

现在我们下载 GenBank identifiers 的列表：

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

现在我们使用这些 GIs 来下载 GenBank records ——注意在老的 Biopython 版本中，你必须将 GI 号用逗号隔开传递给 Entrez，例如在 Biopython 1.59 中，你可以传递一个列表，下面的内容会为你做转换：

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

如果你想看原始的 GenBank 文件，你可以从这个句柄中读取并打印结果：

```
>>> text = handle.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 Austrocy lindropuntia subulata
  ORGANISM Austrocy lindropuntia subulata
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
            Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;
            Caryophyllales; Cactaceae; Opuntioideae; Austrocy lindropuntia.
REFERENCE  1  (bases 1 to 892)
AUTHORS   Butterworth,C.A. and Wallace,R.S.
...
```

在这个例子当中，我们只是得到了原始的记录。为了得到对 Python 友好的格式，我们可以使用 Bio.SeqIO 将 GenBank 数据转化成 SeqRecord 对象，包括 SeqFeature 对象 (请见第 5 章):

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

我们现在可以逐个查看这些 record 来寻找我们感兴趣的信息：

```
>>> for record in records:
>>> ...     print "%s, length %i, with %i features" \
>>> ...         % (record.name, len(record), len(record.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
```

使用这些自动的查询提取功能相对于手动处理是一个很大的进步。尽管这些模块需要遵守 NCBI 每秒钟最多三次的规则，然而 NCBI 有其他像避开高峰时刻的建议。请见章节 9.1。尤其需要注意的是，这个例子没有用到 WebEnv 历史记录特性。你应该使用这个来完成一些琐碎的搜索和下载的工作，请见章节 9.15。

最后，如果你计划重复你的分析，你应该下载这些 record 一次，然后将他们保存在你的硬盘里，在本地进行分析；而不是从 NCBI 下载之后就马上进行分析（像这个例子一样）。

9.14.4 查看物种的谱系关系

仍然以植物为例子，让我们找出 Cyripedioideae 兰花家族的谱系。首先让我们在 Taxonomy 数据库中查找跟 Cyripedioideae 相关的记录，确实找到了一个确切的 NCBI taxonomy 标识号：

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

现在，我们使用 efetch 从 Taxonomy 数据库中下载这些条目，然后解析它：

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

再次，这个 record 保存了许多的信息：

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

我们可以直接从这个 record 获得谱系信息：

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

这个 record 数据包含的信息远远超过在这里显示的——例如查看 "LineageEx" 而不是 "Lineage" 相关的信息，你也可以得到谱系里面的 NCBI taxon 标识号信息。

9.15 使用历史记录和 WebEnv

通常，你想做一系列相关的查询。最典型的是，进行一个搜索，精炼搜索，然后提取详细的搜索结果。你可以通过一系列独立的调用 Entrez 来完成这些工作。然而，NCBI 更希望你利用历史记录支持的优势来完成这个 - 例如将 ESearch 和 EFetch 结合起来。

另外一个关于历史记录支持，典型的使用是结合 EPost 和 EFetch。你可以使用 EPost 来上传一个标识号的列表，这样就开始一些新的 history session。接下来你就可以用 EFetch 指向这个 session 来下载这些数据（而不是那些标识号）。

9.15.1 利用 history 来搜索和下载序列

假设我们想搜索和下载所有的 *Opuntia* rpl16 核酸序列，然后将它们保存到一个 FASTA 文件里。就像章节 9.14.3 里一样，我们可以简单的用 Bio.Entrez.esearch() 得到一个 GI 号的列表，然后调用 Bio.Entrez.efetch() 来下载他们。

然而，被认同的方法是使用历史记录特性来进行搜索。然后，我们可以通过指向这些搜索结果的引用来获取他们 - NCBI 将会提前进行缓冲。

为此，调用 Bio.Entrez.esearch() 是正常的，但是需要额外的 usehistory="y" 参数，

```
>>> from Bio import Entrez
>>> Entrez.email = "history.user@example.com"
>>> search_handle = Entrez.esearch(db="nucleotide",term="Opuntia[orgn] and rpl16",
                                usehistory="y")
>>> search_results = Entrez.read(search_handle)
>>> search_handle.close()
```

当你得到 XML 输出的时候，它仍然包括了常见的搜索结果：

```
>>> gi_list = search_results["IdList"]
>>> count = int(search_results["Count"])
>>> assert count == len(gi_list)
```

然而，你将得到两个额外的信息，WebEnv 会话 cookie 和 QueryKey：

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

将这些值保存到 session_cookie 和 query_key 后，我们可以使用它们作为 Bio.Entrez.efetch() 的参数，而不用提供 GI numbers 的 identifiers。

对于小数据量你一次下载所有的数据也没有关系，但是最好能够分批下载。你可以使用 restart 和 retmax 来说明哪一部分搜索结果是你能得到的（条目以 0 开始计算，返回结果的最大数目）。例如：

```
batch_size = 3
out_handle = 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)
    fetch_handle = Entrez.efetch(db="nucleotide", rettype="fasta", retmode="text",
                                retstart=start, retmax=batch_size,
                                webenv=webenv, query_key=query_key)

    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.close()
```

我们以此为例来说明，这个例子分三次来下载 FASTA records。除非你是要下载基因组或者染色体数目，你最好选取一个比较大的 batch 大小。

9.15.2 利用 history 来搜索和下载综述

这是另外一个 history 的例子，搜索过去几年当中发表的关于 *Opuntia* 的文章，然后下载到一个 MedLine 格式的文件里：

```
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
out_handle = 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)
    fetch_handle = Entrez.efetch(db="pubmed",
                                rettype="medline", retmode="text",
                                retstart=start, retmax=batch_size,
                                webenv=search_results["WebEnv"],
                                query_key=search_results["QueryKey"])

    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.close()
```

在写这份文档的时候，这个搜索返回了 28 个匹配结果 - 但是因为这个是跟时间相关的搜索，因此返回结果会发生变化。像在上面 9.12.1 讲到的一样，你可以使用 Bio.Medline 来解析保存下来的记录。

9.15.3 搜索引用文章

回到 Section 9.7 我们提到可以使用 ELink 来搜索制定文章的引用。不幸的是，这个只包含 PubMed Central (为 PubMed 中所有文献来做这个事情，意味这 NIH 将要付出更多的工作) 包含的那些杂志。让我们以 Biopython PDB parser 文章为例来试试看，PubMed ID 14630660：

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

好极了 - 11 篇文章。但是为什么没有 Biopython 应用笔记 (PubMed ID 19304878) 呢？好吧，你可能已经从变量的名称中猜到了，实际上他们不是 PubMed IDs，而是 PubMed Central IDs。我们的应用笔记是列表当中第三个引用的文章，PMCID 2682512。

那么，如果（像我）你希望得到的是 PubMed IDs 的列表的话，该怎么做呢？好吧，你可以使用再次使用 ELink 来更改他们。这将成为两步处理，所以你应该使用历史记录特性来完成这个工作（章节 9.15）。

但是首先，让我们使用更直接的方法来进行第二次调用 ELink：

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

这次，你可以立即看到 Biopython 应用笔记作为第三个 hit (PubMed ID 19304878)。

现在，让我们重新使用历史记录再试一遍… *TODO*.

最终，不要忘记在 Entrez 调用的时候，加上你 自己 的电子邮箱地址。

第 10 章 SWISS-PROT 和 EXPASY

10.1 解析 Swiss-Prot 文件

Swiss-Prot (<http://www.expasy.org/sprot>) 是一个蛋白质序列数据库。Biopython 能够解析纯文本的 Swiss-Prot 文件, 这种格式也被 Swiss-Prot、TrEMBL 和 PIRPSD 的 UniProt 数据库使用。然而我们并不支持 UniProKB 的 XML 格式文件。

10.1.1 解析 Swiss-Prot 记录

在 5.3.2 章节中, 我们描述过怎样将一个 Swiss-Prot 记录中的序列提出来作为一个 SeqRecord 对象。此外, 你可以将 Swiss-Prot 记录存到 Bio.SwissProt.Record 对象, 这实际上存储了 Swiss-Prot 记录中所包含的全部信息。在这部分我们将介绍怎样从一个 Swiss-Prot 文件中提取 Bio.SwissProt.Record 对象。

为了解析 Swiss-Prot 记录, 我们首先需要得到一个 Swiss-Prot 记录文件。根据该 Swiss-Prot 记录的储存位置和储存方式, 获取该记录文件的方式也有所不同:

- 本地打开 Swiss-Prot 文件:

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

- 打开使用 gzip 压缩的 Swiss-Prot 文件:

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

- 在线打开 Swiss-Prot 文件:

```
>>> import urllib
>>> handle = urllib.urlopen("http://www.somelocation.org/data/someswissprotfile.dat")
```

- 从 ExPASy 数据库在线打开 Swiss-Prot 文件 (见 10.5.1 章节):

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

对于解析来说, 关键点在于 Swiss-Prot 格式的数据, 而不是获取它的方式。

我们可以用 5.3.2 章节中描述的方式, 通过 Bio.SeqIO 来获取格式未知的 SeqRecord 对象。此外, 我们也可以用 Bio.SwissProt 来获取更加匹配基本文件格式的 Bio.SwissProt.Record 对象。

我们使用 read() 函数来从文件中读取一个 Swiss-Prot 记录:

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


该函数只适用于仅存储了一个 Swiss-Prot 记录的文件，而当文件中没有或存在多个记录时使用该函数，会出现 `ValueError` 提示。

现在我们可以输出一些与这些记录相关的信息：

```
>>> print record.description
'RecName: Full=Chalcone synthase 3; EC=2.3.1.74; AltName: Full=Naringenin-chalcone synthase 3;'
>>> for ref in record.references:
...     print "authors:", ref.authors
...     print "title:", ref.title
...
authors: Liew C.F., Lim S.H., Loh C.S., Goh C.J.;
title: "Molecular cloning and sequence analysis of chalcone synthase cDNAs of
Bromheadia finlaysoniana.";
>>> print record.organism_classification
['Eukaryota', 'Viridiplantae', 'Streptophyta', 'Embryophyta', ..., 'Bromheadia']
```

为了解析包含多个 Swiss-Prot 记录的文件，我们使用 `parse` 函数。这个函数能够让我们对文件中的记录进行循环迭代操作。

比如，我们要解析整个 Swiss-Prot 数据库并且收集所有的描述。你可以从 [ExPAsy FTP site](#) 下载这些 gzip 压缩文件 `uniprot_sprot.dat.gz` (大约 300MB)。文件中含有 `uniprot_sprot.dat` 一个文件 (至少 1.5GB)。

如同这一部分刚开始所描述的，你可以按照如下所示的方法使用 python 的 `gzip` 模块打开并解压 `.gz` 文件：

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

然而，解压一个大文件比较耗时，而且每次用这种方式打开一个文件都是比较慢的。所以，如果你有空闲的硬盘空间并且在最开始就在硬盘里通过解压到来得到 `uniprot_sprot.dat`，这样能够在以后就可以像平常那样来打开文件：

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

到 2009 年 6 月为止，从 ExPASy 下载下来的整个 Swiss-Prot 数据库一共有 468851 个 Swiss-Prot 记录，一种建立关于这些记录的描述列表的间接方式就是使用一种列表解析：

```
>>> 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;']
```

或者对记录迭代器使用 `for` 循环：

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

```
>>> len(descriptions)
468851
```

由于输入文件太大，这两种方法在我的新台式机上花费大约十一分钟（用解压好的 uniprot_sprot.dat 作为输入文件）。

从 Swiss-Prot 记录中提取任何你想要的信息也同样简单。比如你想看看一个 Swiss-Prot 记录中的成员，就输入：

```
>>> 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']
```

10.1.2 解析 Swiss-Prot 关键词和分类列表

Swiss-Prot 也会提供一个 keywlist.txt 文件，该文件列出了 Swiss-Prot 中所用到的关键词和分类。其中所包含的词条形式如下：

```
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.
...
```

文件中的词条可以通过使用 Bio.SwissProt.KeyWList 模块中的 parse 函数来解析，并且每一个词条都会被存储在名为 Bio.SwissProt.KeyWList.Record 的 python 字典里。

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

这些命令行将会输出：

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.

...

10.2 解析 Prosite 记录

Prosite 是一个包含了蛋白质结构域、蛋白家族、功能位点以及识别它们的模式和图谱，而且它是和 Swiss-Prot 同时开发出来的。在 Biopython 中，Prosite 记录是由 `Bio.ExPASy.Prosite.Record` 类来表示的，其中的成员与该 Prosite 记录中的不同区域相对应。

一般来说，一个 Prosite 文件可以包含多个 Prosite 记录。比如，从 [ExPASy FTP site](#) 网站下载下来的、容纳了整个 Prosite 记录的 `prosite.dat` 文件，含有 2073 条记录（2007 年 12 月发布的第 20.24 版本）。为了解析这样一个文件，我们再次使用一个迭代器：

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

现在我们可以逐个提取这些记录并输出其中一些信息。比如，使用包含整个 Prosite 数据库的文件将会使我们找到如下等信息：

```
>>> from Bio.ExPASy import Prosite
>>> handle = open("prosite.dat")
>>> records = Prosite.parse(handle)
>>> record = records.next()
>>> record.accession
'PS00001'
>>> record.name
'ASN_GLYCOSYLATION'
>>> record.pdoc
'PDOC00001'
>>> record = records.next()
>>> record.accession
'PS00004'
>>> record.name
'CAMP_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00004'
>>> record = records.next()
>>> record.accession
'PS00005'
>>> record.name
'PKC_PHOSPHO_SITE'
>>> record.pdoc
'PDOC00005'
```

如果你想知道有多少条 Prosite 记录，你可以输入：

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

```
>>> print n
2073
```

为了从这些数据中读取某一条特定的记录，可以使用 `read` 函数：

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

如果并不存在或存在多个你想要找的 Prosite 记录时，这个函数将会输出一个“ValueError”提示。

10.3 解析 Prosite 文件记录

在上述的 Prosite 示例中，像 'PD0C00001'、'PD0C00004'、'PD0C00005' 等这样的编号指的就是 Prosite 文件。Prosite 文件记录可以以单个文件 (`prosite.doc`) 的形式从 ExPASy 获取，并且该文件包含了所有 Prosite 文档记录。

我们使用 `Bio.ExPASy.Prodoc` 中的解析器来解析这些 Prosite 文档记录。比如，为了生成一个包含所有 Prosite 文档记录的编号列表，你可以使用：

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

进一步可以使用 `read()` 函数来对这些数据中具体某一条文档记录来进行查询。

10.4 解析酶记录

ExPASy 的酶数据库是一个关于酶的系统命名信息的数据库。如下所示是一个比较典型的酶的记录

```
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; PD0C00110;
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 ;
//
```

在这个例子中，第一行显示了脂蛋白脂肪酶（第二行）的酶编号 (EC, Enzyme Commission)。脂蛋白脂肪酶其他的名称有“清除因子脂肪酶”和“甘油二酯脂肪酶”（第三行至第五行）。开头为“CA”的那一行显示了该酶的催化活性。评论行开头为“CC”。“PR”行显示了对应 Prosite 文档记录的参考，以及“DR”行显示了 Swiss-Prot 记录的参考。然而并不是所有的词条都必需出现在酶记录当中。

在 Biopython 中，一个酶记录由 `Bio.ExPASy.Enzyme.Record` 类来代表。这个记录源于对应于酶相关文件中所用到的双字母编码的 python 字典和哈希键。为了阅读含有一个酶记录的酶文件，你可以使用 `Bio.ExPASy.Enzyme` 中的 `read` 函数：

```
>>> from Bio.ExPASy import Enzyme
>>> handle = open("lipoprotein.txt")
>>> 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"]
['PD0C00110']

>>> 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'],
 ['046647', 'LIPL_MUSVI'], ['P49060', 'LIPL_PAPAN'], ['P49923', 'LIPL_PIG'],
 ['Q06000', 'LIPL_RAT'], ['Q29524', 'LIPL_SHEEP']]
```

如果没有找到或者找到多个酶记录时，`read` 函数会反馈一个 `ValueError` 提示。

所有酶记录都可以从 [ExPASy FTP site](http://www.expasy.org) 网站下载为单个文件 (`enzyme.dat`)，该文件包含了 4877 个记录 (2009 年 3 月发布的第三版)。为了打开含有多个酶记录的文件，你可以使用 `Bio.ExPASy.Enzyme` 中的 `parse` 函数来获得一个迭代器：

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

我们现在每次都可以对这些记录进行迭代。比如我们可以对那些已有的酶记录做一个 EC 编号列表：

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

10.5 Accessing the ExPASy server

Swiss-Prot、Prosite 和 Prosite 文档记录可以从 <http://www.expasy.org> 的 ExPASy 网络服务器下载到。在 ExPASy 服务器上可以进行六种查询：

`get_prodoc_entry` 下载一个 HTML 格式的 Prosite 文档记录

`get_prosite_entry` 下载一个 HTML 格式的 Prosite 记录

`get_prosite_raw` 下载一个原始格式的 Prosite 或 Prosite 文档记录

`get_sprot_raw` 下载一个原始格式的 Swiss-Prot 记录

`sprot_search_ful` 搜索一个 Swiss-Prot 记录

`sprot_search_de` 搜索一个 Swiss-Prot 记录

为了从 python 脚本来访问该网络服务器，我们可以使用 `Bio.ExPASy` 模块。

10.5.1 获取一个 Swiss-Prot 记录

现在让我们来寻找一个关于兰花的查儿酮合成酶（对于寻找和兰花相关的有趣东西的理由请看 2.3 章节）。查儿酮合成酶参与了植物中类黄酮的生物合成，类黄酮能够合成包含色素和 UV 保护分子等物质。

如果你要对 Swiss-Prot 进行搜索，你可以找到三个关于查儿酮合成酶的兰花蛋白，id 编号为 O23729, O23730, O23731。现在我们要写一个能够获取这些蛋白并能够找到一些有趣的信息的脚本。

首先，我们使用 Bio.ExPASy 中的 `get_sprot_raw()` 函数来获取这些记录。这个函数非常棒，因为你可以给它提供一个 id 然后得到一个原始文本记录（不会受到 HTML 的干扰）。然后我们可以使用 `Bio.SwissProt.read` 来提取对应的 Swiss-Prot 记录，也可以使用 `Bio.SeqIO.read` 来得到一个序列记录 `SeqRecord`。下列代码能够实现我刚刚提到的任务：

```
>>> 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)
```

如果你提供给 `ExPASy.get_sprot_raw` 的编号并不存在，那么 `SwissProt.read(handle)` 会反馈一个 `ValueError` 提示。你可以根据 `ValueException` 异常来找到无效的编号：

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

10.5.2 搜索 Swiss-Prot

现在，你可以察觉到我已经提前知道了这个记录的编号。的确，`get_sprot_raw()` 需要一个词条或者编号。当你并没有编号或者词条的时候，你可使用 `sprot_search_de()` 或者 `sprot_search_ful()` 函数来解决问题。

`sprot_search_de()` 在 ID, DE, GN, OS 和 OG 行进行搜索；`sprot_search_ful()` 则在所有行进行搜索。具体相关细节分别在 <http://www.expasy.org/cgi-bin/sprot-search-de> 和 <http://www.expasy.org/cgi-bin/sprot-search-ful> 上有说明。注意它们的默认情况下并不搜索 TrEMBL（参数为 `trembl`）。还要注意它们返回的是 html 网页，然而编号却可以很容易从中得到：

```
>>> from Bio import ExPASy
>>> import re

>>> handle = ExPASy.sprot_search_de("Orchid Chalcone Synthase")
>>> # or:
>>> # handle = ExPASy.sprot_search_ful("Orchid and {Chalcone Synthase}")
>>> html_results = handle.read()
>>> if "Number of sequences found" in html_results:
...     ids = re.findall(r'HREF="/uniprot/(\w+)"', html_results)
... else:
...     ids = re.findall(r'href="/cgi-bin/niceprot.pl\?(\w+)"', html_results)
```

10.5.3 获取 Prosite 和 Prosite 文档记录

我们可以得到 HTML 格式和原始格式的 Prosite 和 Prosite 文档记录。为了用 biopython 解析 Prosite 和 Prosite 文档记录，你应该使用原始格式的记录。而对于其他的目的，你或许会对 HTML 格式感兴趣。

为了获取一个原始格式的 Prosite 或者 Prosite 文档的记录，请使用 `get_prosite_raw()`。例如，为了下载一个 prosite 记录并以原始格式输出，你可以使用：

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

为了获取一个 Prosite 记录并将其解析成一个 `Bio.Prosite.Record` 对象，请使用：

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

该函数也可以用于获取 Prosite 文档记录并解析到一个 `Bio.ExPASy.Prodoc.Record` 对象：

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

对于不存在的编号，`ExPASy.get_prosite_raw` 返回一个空字符串。当遇到空字符串，`Prosite.read` 和 `Prodoc.read` 会反馈一个 `ValueError` 错误。你可以根据这些错误异常提示来找到无效的编号。

`get_prosite_entry()` 和 `get_prodoc_entry()` 函数可用于下载 HTML 格式的 Prosite 和 Prosite 文档记录。为了生成展示单个 Prosite 记录的网页，你可以使用：

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prosite_entry('PS00001')
>>> html = handle.read()
>>> output = open("myprositerecord.html", "w")
>>> output.write(html)
>>> output.close()
```

类似地，Prosite 文档文本的网页展示如下：

```
>>> from Bio import ExPASy
>>> handle = ExPASy.get_prodoc_entry('PDOC00001')
>>> html = handle.read()
>>> output = open("myprodocrecord.html", "w")
>>> output.write(html)
>>> output.close()
```

对于这些函数，无效的编号会返回一个 HTML 格式的错误信息。

10.6 浏览 Prosite 数据库

[ScanProsite](#) 允许你通过向 Prosite 数据库提供一个 Uniprot 或者 PDB 序列编号或序列来在线浏览蛋白质序列。关于 ScanProsite 更多的信息，请阅读 [ScanProsite 文档](#) 以及 [程序性访问 ScanProsite 说明文档](#)。

你也可以使用 Biopython 的 `Bio.ExPASy.ScanProsite` 模块来从 python 浏览 Prosite 数据库，这个模块既能够帮你安全访问 ScanProsite，也可以对 ScanProsite 返回的结果进行解析。为了查看下边序列中的 Prosite 模式 (pattern)：

```
MEHKEVLLLLLLFLKSGQGEPLDDYVNTQGASLFSVTKKQLGAGSIEECAAKCEEDEEFT
CRAFQYHSKEQQCVIMAENRKSSIIIRMDVVLFEKKVYLSECKTGNGKNYRGTMSTKN
```

你可以使用下边的代码：

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

你可以通过执行 `handle.read()` 获取原始 XML 格式的搜索结果。此外，我们可以使用 `Bio.ExPASy.ScanProsite.read` 来将原始的 XML 数据解析到一个 python 对象：

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

``Bio.ExPASy.ScanProsite.Record`` 对象源自一个由 ScanProsite 返回的包含了 ScanProsite hits 的列表，这个对象也能够存储 hits

```
>>> result.n_seq
1
>>> result.n_match
6
>>> len(result)
6
>>> result[0]
{'signature_ac': u'PS50948', 'level': u'0', 'stop': 98, 'sequence_ac': u'USERSEQ1', 'start': 16, 'score': u'8.873'}
>>> result[1]
{'start': 37, 'stop': 39, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[2]
{'start': 45, 'stop': 48, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00006'}
>>> result[3]
{'start': 60, 'stop': 62, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00005'}
>>> result[4]
{'start': 80, 'stop': 83, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00004'}
>>> result[5]
{'start': 106, 'stop': 111, 'sequence_ac': u'USERSEQ1', 'signature_ac': u'PS00008'}
```

其他的 ScanProsite 参数可以以关键词参数的形式被传递，更多的信息详见 [程序性访问 ScanProsite 说明文档](#)。比如，传递 `lowscore=1` 可以帮助我们找到一个新的低分值 hit：

```
>>> handle = ScanProsite.scan(seq=sequence, lowscore=1)
>>> result = ScanProsite.read(handle)
>>> result.n_match
7
```


第 11 章走向 3D : PDB 模块

Bio.PDB 是 Biopython 中处理生物大分子晶体结构的模块。除了别的类之外，Bio.PDB 包含 PDBParser 类，此类能够产生一个 Structure 对象，以一种较方便的方式获取文件中的原子数据。只是在处理 PDB 文件头所包含的信息时，该类有一定的局限性。

11.1 晶体结构文件的读与写

11.1.1 读取 PDB 文件

首先，我们创建一个 PDBParser 对象：

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

PERMISSIV 标签表示一些与 PDB 文件相关的问题（见 11.7.1）会被忽略（注意某些原子和/或残基会丢失）。如果没有这个标签，则会在解析器运行期间有问题被检测到的时候生成一个 PDBConstructionException 标签。

接着通过 PDBParser 解析 PDB 文件，就产生了 Structure 对象（在此例子中，PDB 文件为 'pdb1fat.ent'，'1fat' 是用户定义的结构名称）：

```
>>> structure_id = "1fat"
>>> filename = "pdb1fat.ent"
>>> s = p.get_structure(structure_id, filename)
```

你可以从 PDBParser 对象中用 get_header 和 get_trailer 方法来提取 PDB 文件中的文件头和文件尾（简单的字符串列表）。然而许多 PDB 文件头包含不完整或错误的信息。许多错误在等价的 mmCIF 格式文件中得到修正。^{*} 因此，如果你对文件头信息感兴趣，可以用下面即将讲到的 MMCIF2Dict 来提取信息，而不用处理 PDB 文件文件头。^{*}

现在澄清了，让我们回到解析 PDB 文件头这件事上。结构对象有个属性叫 header，这是一个将头记录映射到其相应值的 Python 字典。

例子：

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

在这个字典中可用的关键字有 name、head、deposition_date、release_date、structure_method、resolution、structure_reference（映射到一个参考文献列表）、journal_reference、author、和 compound（映射到一个字典，其中包含结晶化合物的各种信息）。

没有创建 Structure 对象的时候，也可以创建这个字典，比如直接从 PDB 文件创建：

```
>>> file = open(filename, 'r')
>>> header_dict = parse_pdb_header(file)
>>> file.close()
```

11.1.2 读取 mmCIF 文件

与 PDB 文件的情形类似，先创建一个 MMCIFParser 对象：

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

然后用这个解析器从 mmCIF 文件创建一个结构对象：

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

为了尽量少访问 mmCIF 文件，可以用 MMCIF2Dict 类创建一个 Python 字典来将所有 mmCIF 文件中各种标签映射到其对应的值上。若有多个值（像 `_atom_site.Cartn_y` 标签，储存的是所有原子的 `*y*` 坐标值），则这个标签映射到一个值列表。从 mmCIF 文件创建字典如下：

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

例：从 mmCIF 文件获取溶剂含量：

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

例：获取包含所有原子 `*y*` 坐标的列表：

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

11.1.3 读取 PDB XML 格式的文件

这个功能暂时还不支持，不过我们确实计划在未来支持这个功能（这项任务并不大）。如果你需要的话联系 Biopython 开发人员 (biopython-dev@biopython.org)。

11.1.4 写 PDB 文件

可以用 PDBIO 类实现。当然也可很方便地输出一个结构的特定部分。

例子：保存一个结构

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

如果你想写出结构的一部分，可以用 *Select* 类（也在 PDBIO 中）来实现。*Select* 有如下四种方法：

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

在默认情况下，每种方法的返回值都为 1（表示 model/chain/residue/atom 被包含在输出结果中）。通过子类化 Select 和返回值 0，你可以从输出中排除 model、chain 等。也许麻烦，但很强大。接下来的代码将只输出甘氨酸残基：

```
>>> 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())
```

如果这部分对你来说太复杂，那么 Dice 模块有一个很方便的 extract 函数，它可以输出一条链中起始和终止氨基酸残基之间的所有氨基酸残基。

11.2 结构的表示

一个 Structure 对象的整体布局遵循称为 SMCRA (Structure/Model/Chain/Residue/Atom，结构/模型/链/残基/原子) 的体系架构：

- 结构由模型组成
- 模型由多条链组成
- 链由残基组成
- 多个原子构成残基

这是很多结构生物学家/生物信息学家看待结构的方法，也是处理结构的一种简单而有效的方法。在需要的时候加上额外的材料。一个 Structure 对象的 UML 图（暂时忘掉 Disordered 吧）如下图所示 11.1。这样的数据结构不一定最适用于表示一个结构的生物大分子内容，但要很好地解释一个描述结构的文件中所呈现的数据（最典型的如 PDB 或 MMCIF 文件），这样的数据结构就是必要的了。如果这种层次结构不能表示一个结构文件的内容，那么可以相当确定是这个文件有错误或至少描述结构不够明确。一旦不能生成 SMCRA 数据结构，就有理由怀疑出了故障。因此，解析 PDB 文件可用于检测可能的故障。我们将在 11.7.1 小节给出关于这一点的一些例子。

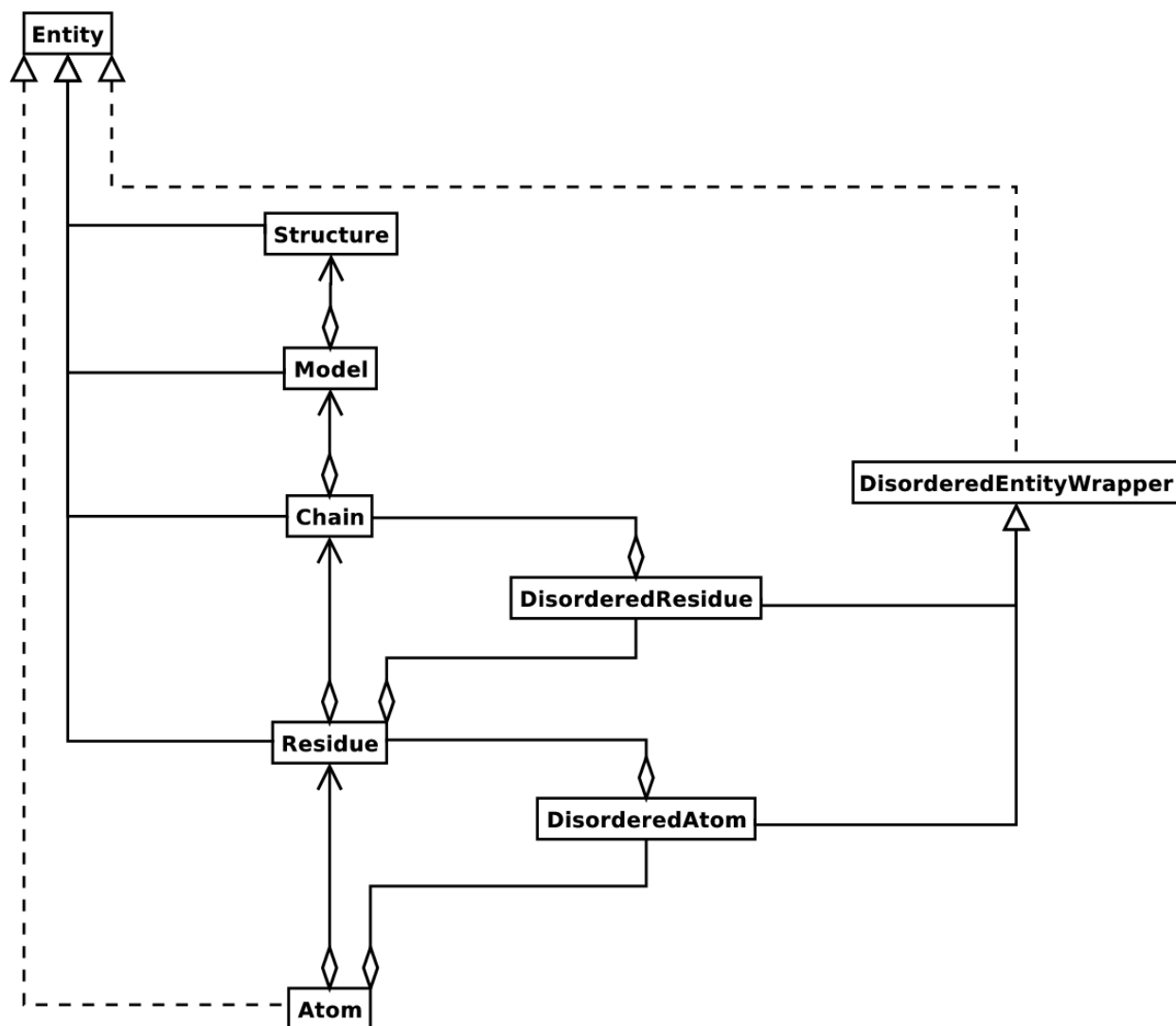


图 11.1：用来表示大分子结构的 `Structure` 类的 SMCRS 体系的 UML 图。带方块的实线表示集合，带箭头的实线表示引用，带三角形的实线表示继承，带三角形的虚线表示接口实现。

结构，模型，链，残基都是实体基类的子类。原子类仅仅（部分）实现了实体接口（因为原子类没有子类）。

对于每个实体子类，你可以用该子类的一个唯一标识符作为键来提取子类（比如，可以用原子名称作为键从残基对象中提取一个原子对象；用链的标识符作为键从域对象中提取链）。

紊乱原子和残基用 `DisorderedAtom` 和 `DisorderedResidue` 类来表示，二者都是 `DisorderedEntityWrapper` 基类的子类。它们隐藏了紊乱的复杂性，表现得与原子和残基对象无二。

一般地，一个实体子类（即原子，残基，链，模型）能通过标识符作为键来从父类（分别为残基，链，模型，结构）中提取。

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

你可以从一个父实体对象获得所有子实体的列表。需要注意的是，这个列表以一种特定的方式排列（例如根据在模型对象中链对象的链标识符来排序）。

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

你也可以从子类得到父类：

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

在 SMCRA 的所有层次水平，你还可以提取一个完整 *id*。完整 *id* 是包含所有从顶层对象（结构）到当前对象的 *id* 的一个元组。一个残基对象的完整 *id* 可以这么得到：

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

这对应于：

- *id* 为“1abc”的结构
- *id* 为 0 的模型
- *id* 为“A”的链
- *id* 为 (“”, 10, “A”) 的残基

这个残基 *id* 表示该残基不是异质残基（也不是水分子），因为其异质值为空；而序列标识符为 10，插入码为“A”。

要得到实体的 *id*，用 `get_id` 方法即可：

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

可以用 `has_id` 方法来检查这个实体是否有子类具有给定 *id*：

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

实体的长度等于其子类的个数：

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

对于从父实体得到的子实体，可以删除，重命名，添加等等，但这并不包含任何完整性检查（比如，有可能添加两个相同 *id* 的残基到同一条链上）。这就真的需要包含完整性检查的装饰类（Decorator）来完成了，但是如果你想使用原始接口的话可以查看源代码（Entity.py）。

11.2.1 结构

结构对象是层次中的最高层。其 *id* 是用户指定的一个字符串。结构包含一系列子模型。大部分晶体结构（但不是全部）含有一个单一模型，但是 NMR 结构通常由若干模型构成。晶体结构中大部分分子的乱序也能导致多个模型。

11.2.2 模型

结构域对象的 *id* 是一个整数，源自该模型在所解析文件中的位置（自动从 0 开始）。晶体结构通常只有一个模型（*id* 为 0），而 NMR 文件通常含有多个模型。然而许多 PDB 解析器都假定只有一个结构域，Bio.PDB 中的 `Structure` 类就设计成能轻松处理含有不止一个模型的 PDB 文件。

举个例子，从一个结构对象中获取其第一个模型：

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

模型对象存储着子链的列表。

11.2.3 链

链对象的 id 来自 PDB/mmCIF 文件中的链标识符，是个单字符（通常是一个字母）。模型中的每个链都具有唯一的 id。例如，从一个模型对象中取出标识符为“A”的链对象：

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

链对象储存着残基对象的列表。

11.2.4 残基

一个残基 id 是一个三元组：

- 异质域 (hetfield)，即：
 - 'W' 代表水分子
 - 'H_' 后面紧跟残基名称，代表其它异质残基（例如 'H_GLC' 表示一个葡萄糖分子）
 - 空值表示标准的氨基酸和核酸

采用这种体制的理由在 11.4.1 部分有叙述。

- 序列标识符 (resseq)，一个描述该残基在链上的位置的整数（如 100）；
- 插入码 (icode)，一个字符串，如“A”。插入码有时用来保存某种特定的、想要的残基编号体制。一个 Ser 80 的插入突变（比如在 Thr 80 和 Asn 81 残基间插入）可能具有如下序列标识符和插入码：Thr 80 A, Ser 80 B, Asn 81。这样一来，残基编号体制保持与野生型结构一致。

因此，上述的葡萄糖残基 id 就是（'H_GLC'，100，'A'）。如果异质标签和插入码为空，那么可以只使用序列标识符：

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

异质标签的起因是许许多多的 PDB 文件使用相同的序列标识符表示一个氨基酸和一个异质残基或一个水分子，这会产生一个很明显的问题，如果不使用异质标签的话。

毫不奇怪，一个残基对象存储着一个子原子集，它还包含一个表示残基名称的字符串（如“ASN”）和残基的片段标识符（这对 X-PLOR 的用户来说很熟悉，但是在 SMCRA 数据结构的构建中没用到）。

让我们来看一些例子。插入码为空的 Asn 10 具有残基 id（'', 10, ''）；Water 10，残基 id（'W', 10, ''）；一个序列标识符为 10 的葡萄糖分子（名称为 GLC 的异质残基），残基 id 为（'H_GLC', 10, ''）。在这种情况下，三个残基（具有相同插入码和序列标识符）可以位于同一条链上，因为它们的残基 id 是不同的。

大多数情况下，hetflag 和插入码均为空，如（'', 10, ''）。在这些情况下，序列标识符可以用作完整 id 的快捷方式：

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

一个链对象中每个残基对象都应该具有唯一的 id。但是对含紊乱原子的残基，要以一种特殊的方式来处理，详见 11.3.3。

一个残基对象还有大量其它方法：

```
>>> 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
```

你可以用 `is_aa(residue)` 来检验一个残基对象是否为氨基酸。

11.2.5 原子

原子对象储存着所有与原子有关的数据，它没有子类。原子的 `id` 就是它的名称（如，“OG”代表 Ser 残基的侧链氧原子）。在残基中原子 `id` 必需是唯一的。此外，对于紊乱原子会产生异常，见 11.3.2 小节的描述。

原子 `id` 就是原子名称（如 ‘CA’）。在实践中，原子名称是从 PDB 文件中原子名称去除所有空格而创建的。

但是在 PDB 文件中，空格可以是原子名称的一部分。通常，钙原子称为 ‘CA.’ 是为了和 $C\alpha$ 原子（叫做 ‘.CA.’）区分开。在这种情况下，如果去掉空格就会产生问题（如统一一个残基中的两个原子都叫做 ‘CA’），所以保留空格。

在 PDB 文件中，一个原子名字由 4 个字符组成，通常头尾皆为空格。为了方便使用，空格通常可以去掉（在 PDB 文件中氨基酸的 $C\alpha$ 原子标记为“.CA.”，点表示空格）。为了生成原子名称（然后是原子 `id`），空格删掉了，除非会在一个残基中造成名字冲突（如两个原子对象有相同的名称和 `id`）。对于后面这种情况，会尝试让原子名称包含空格。这种情况可能会发生在，比如残基包含名称为“.CA.”和“.CA.”的原子，尽管这不太可能。

所存储的原子数据包括原子名称，原子坐标（如果有的话还包括标准差），B 因子（包括各向异性 B 因子和可能存在的标准差），`altloc` 标识符和完整的、包括空格的原子名称。较少用到的项如原子序号和原子电荷（有时在 PDB 文件中规定）也就没有存储。

为了处理原子坐标，可以用 ‘Atom’ 对象的 `transform` 方法。用 `set_coord` 方法可以直接设定原子坐标。

一个 Atom 对象还有如下其它方法：

```
>>> 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.")
```

`siguij`，各向异性 B 因子和 `sigatm` Numpy 阵列可以用来表示原子坐标。

`get_vector` 方法会返回一个代表 Atom 对象坐标的 Vector 对象，可以对原子坐标进行向量运算。Vector 实现了完整的三维向量运算、矩阵乘法（包括左乘和右乘）和一些高级的、与旋转相关的操作。

举个 Bio.PDB 的 Vector 模块功能的例子，假设你要查找 Gly 残基的 $C\beta$ 原子的位置，如果存在的话。将 Gly 残基的 N 原子沿 $C\alpha$ -C 化学键旋转 -120 度，能大致将其放在一个真正的 $C\beta$ 原子的位置上。怎么做呢？就是下面这样使用 Vector 模块中的 “rotaxis” 方法（能用来构造一个绕特定坐标轴的旋转）：

```
# 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 = rotxaxis(-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
```

这个例子展示了在原子数据上能进行一些相当不平凡的向量运算，这些运算会很有用。除了所有常用向量运算（叉积（用 `**`），点积（用 `*`），角度，取范数等）和上述提到的 `rotxaxis` 函数，`Vector` 模块还有方法能旋转（`rotmat`）或反射（`refmat`）一个向量到另外一个向量上。

11.2.6 从结构中提取指定的 Atom/Residue/Chain/Model

举些例子如下：

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

还可以用一个快捷方式：

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

11.3 紊乱

Bio.PDB 能够处理紊乱原子和点突变（比如 Gly 和 Ala 残基在相同位置上）。

11.3.1 一般性方法

紊乱可以从两个角度来解决：原子和残基的角度。一般来说，我们尝试压缩所有由紊乱引起的复杂性。如果你仅仅想遍历所有 $C\alpha$ 原子，那么你不必在意一些具有紊乱侧链的残基。另一方面，应该考虑在数据结构中完整地表示紊乱性。因此，紊乱原子或残基存储在特定的对象中，这些对象表现得就像毫无紊乱。这可以通过表示紊乱原子或残基的子集来完成。至于挑选哪个子集（例如使用 Ser 残基的哪两个紊乱 OG 侧链原子位置），由用户来决定。

11.3.2 紊乱原子

紊乱原子可以用普通的 `Atom` 对象来表示，但是所有表示相同物理原子的 `Atom` 对象都存储在一个 `DisorderedAtom` 对象中（见图 11.1）。`DisorderedAtom` 对象中每个 `Atom` 对象都能用它的 `altloc` 标识符来唯一地索引。`DisorderedAtom` 对象将所有未捕获方法的调用发送给选定的 `Atom` 对象，缺省对象是代表最高使用率的原子的那个。当然用户可以使用其 `altloc` 标识符来更改选定的 `Atom` 对象。以这种方式，原子紊乱就正确地表示出来而没有很多额外的复杂性。换言之，如果你对原子紊乱不感兴趣，你也不会被它困扰。

每个紊乱原子都有一个特征性的 altloc 标识符。你可以设定：一个 DisorderedAtom 对象表现得像与一个指定的 altloc 标识符相关的 Atom 对象：

```
>>> 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"
```

11.3.3 紊乱残基

普通例子

最常见的例子是一个残基包含一个或多个紊乱原子。这显然可以通过用 DisorderedAtom 对象表示这些紊乱原子来解决，并将 DisorderedAtom 对象存储在一个 Residue 对象中，就像正常的 Atom 对象那样。通过将所有未捕获方法调用发送给其中一个 Atom 对象（被选定的 Atom 对象），DisorderedAtom 对象表现完全像一个正常的原子对象（事实上这个原子有最高的使用率）。

点突变

一个特殊的例子就是当紊乱是由点突变导致的时候，也就是说，在晶体结构中出现一条多肽的两或多个点突变。关于这一点，可以在 PDB 结构 1EN2 中找到一个例子。

既然这些残基属于不同的残基类型（举例说 Ser 60 和 Cys 60），那么它们不应该像通常情况一样存储在一个单一 Residue 对象中。这种情况下每个残基用一个 Residue 对象来表示，两种 Residue 对象都保存在一个单一 DisorderedResidue 对象中（见图. 11.1）。

DisorderedResidue 对象将所有未捕获方法发送给选定的 Residue 对象（默认是所添加的最后一个 Residue 对象），因此表现得像一个正常的残基。在 DisorderedResidue 中每个 Residue 对象可通过残基名称来唯一标识。在上述例子中，残基 Ser 60 在 DisorderedResidue 对象中的 id 为“SER”，而残基 Cys 60 则是“CYS”。用户可以通过这个 id 选择在 DisorderedResidue 中的有效 Residue 对象。

例子：假设一个链在位置 10 有一个由 Ser 和 Cys 残基构成的点突变。确信这个链的残基 10 表现为 Cys 残基。

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

另外，通过使用 (Disordered)Residue 对象的 get_unpacked_list 方法，你能获得所有 Atom 对象的列表（也就是说，所有 DisorderedAtom 对象解包到它们各自的 Atom 对象）。

11.4 异质残基

11.4.1 相关问题

关于异质残基的一个很普遍的问题是同一条链中的若干异质和非异质残基有同样的序列标识符（和插入码）。因此，要为每个异质残基生成唯一的 id，水分子和其他异质残基应该以不同的方式来对待。

记住 Residue 残基有一个元组 (hetfield, resseq, icode) 作为 id。hetfield 值为空 (“”) 表示为氨基酸和核酸；为一个字符串，则表示水分子和其他异质残基。hetfield 的内容将在下面解释。

11.4.2 水残基

水残基的 hetfield 字符串由字母“W”构成。所以水分子的一个典型的残基 id 为 (“W”, 1, “”)。

11.4.3 其他异质残基

其他异质残基的 hetfield 字符以“H.”起始，后接残基名称。一个葡萄糖分子，比如残基名称为“GLC”，则 hetfield 字符为“H.GLC”；它的残基 id 可以是 (“H.GLC”, 1, “”)。

11.5 浏览 Structure 对象

解析 PDB 文件，提取一些 Model、Chain、Residue 和 Atom 对象

```
>>> 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"]
```

迭代遍历一个结构中的所有原子

```
>>> 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
...
...
```

有个快捷方式可以遍历一个结构中所有原子：

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

类似地，遍历一条链中的所有原子，可以这么做：

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

遍历模型中的所有残基

或者，如果你想遍历在一条模型中的所有残基：

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

你也可以用 `Selection.unfold_entities` 函数来获取一个结构的所有残基：

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

或者获得链上的所有原子：

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

明显的是，A=atom, R=residue, C=chain, M=model, S=structure。你可以用这种标记返回层次中的上层，如从一个 Atoms 列表得到（唯一的）Residue 或 Chain 父类的列表：

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

更多信息详见 API 文档。

从链中提取异质残基（如 resseq 10 的葡萄糖（GLC）部分）

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

打印链中所有异质残基

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

输出一个结构分子中所有 B 因子大于 50 的 CA 原子的坐标

```
>>> 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()
...
```

输出所有含紊乱原子的残基

```
>>> 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
... 
```

遍历所有紊乱原子，并选取所有具有 **altloc A** 的原子（如果有的话）

这将会保证，SMCRA 数据结构会表现得如同只存在 altloc A 原子一样。

```
>>> 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")
... 
```

从 Structure 对象中提取多肽

为了从一个结构中提取多肽，需要用 PolypeptideBuilder 从 Structure 构建一个 Polypeptide 对象的列表，如下所示：

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

Polypeptide 对象正是 Residue 对象的一个 UserList，总是从单结构域（在此例中为模型 1）中创建而来。你可以用所得 Polypeptide 对象来获取序列作为 Seq 对象，或获得 $C\alpha$ 原子的列表。多肽可以通过一个 C-N 化学键或一个 $C\alpha$ - $C\alpha$ 化学键距离标准来建立。

例子：

```
# 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()
... 
```

需要注意的是，上例中通过 PolypeptideBuilder 只考虑了结构的模型 0。尽管如此，还是可以用 PolypeptideBuilder 从 Model 和 Chain 对象创建 Polypeptide 对象。

获取结构的序列

要做的第一件事就是从结构中提取所有多肽（如上所述）。然后每条多肽的序列就容易从 `Polypeptide` 对象获得。该序列表示为一个 `Biopython Seq` 对象，它的字母表由 `ProteinAlphabet` 对象来定义。

例子：

```
>>> seq = polypeptide.get_sequence()
>>> print seq
Seq('SNVVE...', <class Bio.Alphabet.ProteinAlphabet>)
```

11.6 分析结构

11.6.1 度量距离

重载原子的减法运算来返回两个原子之间的距离。

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

11.6.2 度量角度

用原子坐标的向量表示，和 `Vector` 模块中的 `calc_angle` 函数可以计算角度。

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

11.6.3 度量扭转角

用原子坐标的向量表示，然后用 `Vector` 模块中的 `calc_dihedral` 函数可以计算角度。

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

11.6.4 确定原子-原子触点

用 `NeighborSearch` 来进行邻接查询。用 C 语言写的（使得运行很快）KD 树模块（见 `Bio.KDTree`）可以用来完成邻接查询。它也包含了一个快速方法来找出相距一定距离的所有点对。

11.6.5 叠加两个结构

可以用 `Superimposer` 对象将两个坐标集叠加。这个对象计算出旋转和平移矩阵，该矩阵旋转两个列表上相重叠的原子使其满足 RMSD 最小。当然这两个列表含有相同数目的原子。`Superimposer` 对象也可以将旋转/平移应用在一列原子上。旋转和平移作为一个元组储存在 `Superimposer` 对象的 `rotran` 属性中（注意，旋转是右乘），RMSD 储存在属性 `rmsd` 中。

`Superimposer` 使用的算法来自 [17 , Golub & Van Loan] 并使用了奇异值分解（这是通用 `Bio.SVDSuperimposer` 模块中实现了的）。

例子：

```
>>> 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.rmsd
# Apply rotation/translation to the moving atoms
>>> sup.apply(moving)
```

为了基于有效位点来叠加两个结构，用有效位点的原子来计算旋转/平移矩阵（如上所述），并应用到整个分子。

11.6.6 双向映射两个相关结构的残基

首先，创建一个 FASTA 格式的比对文件，然后使用“`StructureAlignment`”类。这个类也可以用来比对两个以上的结构。

11.6.7 计算半球暴露（HSE）

半球暴露（Half Sphere Exposure, HSE）是对溶剂暴露 [20] 的一种新的二维度量。根本上，它计数了围绕一个残基，在其侧链方向上及反方向（在 13 围内）的 $C\alpha$ 原子。尽管简单，它表现得比溶剂暴露的其它度量都要好。

HSE 有两种风味： $HSE\alpha$ 和 $HSE\beta$ 。前者仅用到 $C\alpha$ 原子的位置，而后者用到 $C\alpha$ 和 $C\beta$ 原子的位置。HSE 度量是由 `HSExposure` 类来计算的，这个类也能计算触点数目。后一个类有方法能返回一个字典，该字典将一个“`Residue`”对象映射到相应的 $HSE\alpha$, $HSE\beta$ 和触点数目值。

例子：

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

11.6.8 确定二级结构

为了这个功能，你需要安装 DSSP（并获得一个对学术性使用免费的证书，参见 <http://www.cmbi.kun.nl/gv/dssp/>）。然后用 DSSP 类，可以映射 Residue 对象到其二级结构上（和溶剂可及表面区域）。DSSP 代码如下表所列表 11.1。注意 DSSP（程序及其相应的类）不能处理多个模型！

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

Table 11.1: Bio.PDB 中的 DSSP 代码。

DSSP 类也可以用来计算残基的溶剂可及表面。还请参考 11.6.9。

11.6.9 计算残基深度

残基深度是残基原子到溶剂可及表面的平均距离。它是溶剂可及性的一种相当新颖和非常强大的参数化。为了这个功能，你需要安装 Michel Sanner 的 MSMS 程序 (http://www.scripps.edu/pub/olson-web/people/sanner/html/msms_home.html)。然后使用 ResidueDepth 类。这个类像字典一样将 Residue 对象映射到相应的（残基深度， $C\alpha$ 深度）元组。 $C\alpha$ 深度是残基的 $C\alpha$ 原子到溶剂可及表面的距离。

例子：

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

你也可以以带有表面点的数值 Python 数组的形式获得分子表面本身（通过 `get_surface` 函数）。

11.7 PDB 文件中的常见问题

众所周知，很多 PDB 文件包含语义错误（不是结构本身的错误，而是在 PDB 文件中的表示）。Bio.PDB 可以有两种方式来处理这个问题。PDBParser 对象能表现出两种方式：严格方式和宽容方式（默认方式）：

例子：

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

在宽容状态（默认），明显包含错误的 PDB 文件会被“纠正”（比如说一些残基或原子丢失）。这些错误包括：

- 多个残基使用同一个标识符
- 多个原子使用同一个标识符（考虑 altloc 标识符）

这些错误暗示了 PDB 文件中确实存在错误（详情见 [18, Hamelryck and Manderick, 2003]）。在严格模式，带错的 PDB 文件会引发异常，这有助于发现 PDB 文件中的错误。

但是有些错误能自动修正。正常情况下，每个紊乱原子应该会有一个非空 altloc 标识符。可是很多结构没有遵循这个惯例，而在同一原子的两个紊乱位置存在一个空的和一个非空的标识符。这个错误会被以正确的方式自动解析。

有时候一个结构会有这样的情况：一部分残基属于 A 链，接下来一部分残基属于 B 链，然后又有一部分残基属于 A 链，也就是说，这种链是“断的”。这也能被自动正确解析。

11.7.1 例子

PDBParser/Structure 类经过了将近 800 个结构（每个都属于不同的 SCOP 超家族）上的测试。测试总共耗时 20 分钟左右，或者说平均每个结构只需 1.5 秒。在一台 1000 MHz 的 PC 上只需 10 秒就可解析包含近 64000 个原子的大核糖体亚基（1FFK）的结构。

当不能建立明确的数据结构时会发生三类异常。在这三类异常中，可能的起因是 PDB 文件中一个本应修正的错误。这些情况下产生异常要比冒险地错误描述一个数据结构中的结构好得多。

11.7.1.1 重复残基

一个结构包含在一条链中具有相同的序列标识符（resseq 3）和 icode 的两个氨基酸残基。仔细观察可以发现这条链包含残基：Thr A3, ..., Gly A202, Leu A3, Glu A204。很明显第二个 Leu A3 应该是 Leu A203。类似的情况也存在于结构 1FFK（比如它包含残基 Gly B64, Met B65, Glu B65, Thr B67，也就是说 Glu B65 应该是 Glu B66）上。

11.7.1.2 重复原子

结构 1EJG 含有在 A 链 22 位的一个 Ser/Pro 点突变。依次，Ser 22 含一些紊乱原子。和期望的一样，所有属于 Ser 22 的原子都有一个非空的 altloc 标识符（B 或 C）。所有 Pro 22 的原子都有 altloc A，除了含空 altloc 的 N 原子。这会生成一个异常，因为一个点突变处属于两个残基的所有原子都应该有非空的 altloc。结果这个原子很可能被 Ser 和 Pro 22 共用，而 Ser22 丢失了这个 N 原子。此外，这也点出了文件中的一个问题：这个 N 原子应该出现在 Ser 和 Pro 残基中，两种情形下都与合适的 altloc 标识符关联。

11.7.2 自动纠正

一些错误相当普遍且能够在没有太大误解风险的情况下容易地纠正过来。这些错误列在下面。

11.7.2.1 紊乱原子的空 altloc

正常情况下，每个紊乱原子应该会有一个非空 altloc 标识符，可是很多结构没有遵循这个惯例，而是在同一原子的两个紊乱位置存在一个空的和一个非空的标识符。这个错误会被以正确的方式自动解析。

11.7.2.2 断链

有时候一个结构会有这样的情况：一部分残基属于 A 链，接下来一部分残基属于 B 链，然后又有一部分残基属于 A 链，也就是说，链是“断的”，这也能被正确的解析。

11.7.3 致命错误

有时候一个 PDB 文件不能被明确解释。这会产生异常并等待用户去修正这个 PDB 文件，而不是猜测和冒出错的风险。这些异常列在下面。

11.7.3.1 重复残基

在一条链上的所有残基都应该有一个唯一的 id。该 id 基于下述生成：

- 序列标识符 (resseq)
- 插入码 (icode)
- hetfield 字符 (“W”代表水，“H.”后面的残基名称代表其他异质残基)
- 发生点突变的残基的名称 (在 DisorderedResidue 对象中存储 Residue 对象)

如果这样还不能生成一个唯一的 id，那么肯定是一些地方出了错，这时会生成一个异常。

11.7.3.2 重复原子

一个残基上所有原子应该有一个唯一的 id，这个 id 基于下述产生：

- 原子名称 (不带空格，否则会报错)
- altloc 标识符

如果这样还不能生成一个唯一的 id，那么肯定是一些地方出了错，这时会生成一个异常。

11.8 访问 Protein Data Bank

11.8.1 从 Protein Data Bank 下载结构

结构可以从 PDB (Protein Data Bank) 通过 PDBList 对象的 retrieve_pdb_file 方法下载。这种方法的要点是结构的 PDB 标识符。

```
>>> pdbl = PDBList()
>>> pdbl.retrieve_pdb_file('1FAT')
```

PDBList 类也能用作命令行工具：

```
python PDBList.py 1fat
```

下载的文件将以 pdb1fat.ent 为名保存在当前工作目录。注意 retrieve_pdb_file 方法还有个可选参数 pdir 用来指定一个特定的路径来保存所下载的文件。

retrieve_pdb_file 方法还有其他选项可以指定下载所用的压缩格式 (默认的 .Z 格式和 gunzip 格式)。另外，在创建 PDBList 对象时还可以指定 PDB ftp 站点。默认使用 Worldwide Protein Data Bank (<ftp://ftp.wwpdb.org/pub/pdb/data/structures/divided/pdb/>)。详细内容参见 API 文档。再次感谢 Kristian Rother 对此模块的所做的贡献。

11.8.2 下载整个 PDB

下面的命令将会保存所有 PDB 文件至 `/data/pdb` 目录：

```
python PDBList.py all /data/pdb
```

```
python PDBList.py all /data/pdb -d
```

在 API 中这个方法叫做 `download_entire_pdb`。添加 `-d` 会在同一目录下保存所有文件。否则将分别保存至 PDB 风格的、与其 PDB ID 对应的子目录中。根据网速，完整的下载全部 PDB 文件大概需要 2-4 天。

11.8.3 保持本地 PDB 拷贝的更新

这也能通过 `PDBList` 对象来完成。可以简单的创建一个 `PDBList` 对象（指定本地 PDB 拷贝的目录），然后调用 `update_pdb` 方法：

```
>>> pl = PDBList(pdb='/data/pdb')
>>> pl.update_pdb()
```

当然还可以每周用 `cronjob` 实现本地拷贝自动更新。还可以指定 PDB ftp 站点（详见 API 文档）。

`PDBList` 有其他许多其它方法可供调用。`get_all_obsolete` 方法可以获取所有已经废弃的 PDB 项的一个列表；`changed_this_week` 方法可以用于获得当前一周内新增加、修改或废弃的 PDB 项。更多 `PDBList` 的用法参见 API 文档。

11.9 常见问题

11.9.1 Bio.PDB 测试得如何？

事实上，相当好。Bio.PDB 已经在从 PDB 获得的近 5500 个结构上广泛的测试过，所有结构都能正确地解析。更多细节可以参考在 Bioinformatics 上发表的关于 Bio.PDB 的文章。作为一个可靠的工具，Bio.PDB 已经并正用于许多研究项目中。我几乎每天都在用它，出于研究目的、提升其性能和增加新属性。

11.9.2 它有多快？

`PDBParser` 的性能经过将近 800 个结构测试（每个都属于不同的 SCOP 超家族），总共花费 20 分钟左右，也就是说平均每个结构只需 1.5 秒。在一台 1000 MHz 的 PC 上解析巨大的包含近 64000 个原子的核糖体亚单位 (1FKK) 只需 10 秒。总而言之，它比很多应用程序都快得多。

11.9.3 是否支持分子图形展示？

不直接支持，很大程度上是因为已有相当多基于 Python 或 Python-aware 的解决方案，也可能会用到 Bio.PDB。顺便说一下，我的选择是 Pymol（我在 Pymol 中使用 Bio.PDB 非常成功，将来 Bio.PDB 中会有特定的 PyMol 模块）。基于 Python 或 Python-aware 的分子图形解决方案包括：

- PyMol: <http://pymol.sourceforge.net/>
- Chimera: <http://www.cgl.ucsf.edu/chimera/>
- PMV: <http://www.scripps.edu/~sanner/python/>

- Coot: <http://www.ysbl.york.ac.uk/~emsley/coot/>
- CCP4mg: <http://www.ysbl.york.ac.uk/~lizp/molgraphics.html>
- mmLib: <http://pymmlib.sourceforge.net/>
- VMD: <http://www.ks.uiuc.edu/Research/vmd/>
- MMTK: <http://starship.python.net/crew/hinsen/MMTK/>

11.9.4 谁在用 Bio.PDB ?

Bio.PDB 曾用于构建 DISEMBL, 一个能预测蛋白结构中的紊乱区域的 web 服务器 (<http://dis.embl.de/>) ; COLUMBA, 一个提供注释过的蛋白结构的站点 (<http://www.columba-db.de/>)。Bio.PDB 也用于进行 PDB 中蛋白质间有效位点的大规模相似性搜索 [19, Hamelryck, 2003], 用于开发新的算法来鉴别线性二级结构元件 [26, Majumdar *et al.*, 2005]。

从对特征和信息的需求判断, 许多大型制药公司也使用 Bio.PDB。

第 12 章 BIO.POPGEN：群体遗传学

Bio.PopGen 是一个群体遗传学相关的模块，在 Biopython 1.44 及以后的版本中可用。

该模块的中期目标是支持各种类型的数据格式、应用程序和数据库。目前，该模块正在紧张的开发中，并会快速实现对新特征的支持。这可能会带来一些不稳定的 API，尤其是当你使用的是开发版。不过，我们正式公开发行的 API 应该更加稳定。

12.1 GenePop

GenePop (<http://genepop.curtin.edu.au/>) 是一款主流的群体遗传学软件包，支持 Hardy-Weinberg 检验、连锁不平衡、群体分化、基础统计计算、 F_{st} 和迁移率估计等等。GenePop 并不支持基于序列的统计计算，因为它并不能处理序列数据。GenePop 文件格式广泛用于多种其它的群体遗传学应用软件，因此成为群体遗传学领域重要格式。

Bio.PopGen 提供 GenePop 文件格式解析器和生成器，同时也提供操作记录内容的小工具。此处有个关于怎样读取 GenePop 文件的示例（你可以在 Biopython 的 Test/PopGen 文件夹下找到 GenePop 示例文件）：

```
from Bio.PopGen import GenePop

handle = open("example.gen")
rec = GenePop.read(handle)
handle.close()
```

它将读取名为 example.gen 的文件并解析。如果你输出 rec，那么该记录将会以 GenePop 格式再次输出。

在 rec 中最重要的信息是基因座名称和群体信息（当然不止这些，请使用 help(GenePop.Record) 获得 API 帮助文档）。基因座名称可以在 rec.loci_list 中找到，群体信息可以在 rec.populations 中找到。群体信息是一个列表，每个群体（population）作为其中一个元素。每个元素本身又是包含个体（individual）的列表，每个个体包含个体名和等位基因列表（每个 marker 两个元素），下面是一个 rec.populations 的示例：

```
[
  [
    ('Ind1', [(1, 2), (3, 3), (200, 201)]),
    ('Ind2', [(2, None), (3, 3), (None, None)]),
  ],
  [
    ('Other1', [(1, 1), (4, 3), (200, 200)]),
  ]
]
```

在上面的例子中，我们有两个群体，第一个群体包含两个个体，第二个群体只包含一个个体。第一个群体的第一个个体叫做 Ind1，紧接着是 3 个基因座各自的等位基因信息。请注意，对于任何的基因座，信息可以缺失（如上述个体 Ind2）。

有几个可用的工具函数可以处理 GenePop 记录，如下例：

```
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 不支持群体名，这种限制有时会很麻烦。Biopython 对群体名的支持正在规划中，这些功能扩展仍会保持对标准格式的兼容性。同时，中期目标是对 GenePop 网络服务的支持。

12.2 溯祖模拟 (Coalescent simulation)

溯祖模拟是一种对群体遗传学信息根据时间向后推算的模型 (backward model)。对祖先的模拟是通过寻找到最近共同祖先 (Most Recent Common Ancestor, MRCA) 完成。从 MRCA 到目前这一代样本间的血统关系有时称为家系 (genealogy)。简单的情况是假定群体大小固定，单倍型，无群体结构，然后模拟无选择压的单个基因座的等位基因。

溯祖理论被广泛用于多种领域，如选择压力检测、真实群体的群体参数估计以及疾病基因图谱。

Biopython 对溯祖的实现不是去创建一个新的内置模拟器，而是利用现有的 SIMCOAL2 (<http://cmpg.unibe.ch/software/simcoal2/>)。与其他相比，SIMCOAL2 允许存在群体结构，多群体事件，多种类型的可发生重组的基因座 (SNPs, 序列, STRs/微卫星和 RFLPs)，具有多染色体的二倍体

和测量偏倚 (ascertainment bias)。注意，SIMCOAL2 并不支持所有的选择模型。建议阅读上述链接中的 SIMCOAL2 帮助文档。

SIMCOAL2 的输入是一个指定所需的群体和基因组的文件，输出是一系列文件（通常在 1000 个左右），它们包括每个亚群 (subpopulation) 中模拟的个体基因组。这些文件有多种途径，如计算某些统计数据 (e.g. F_{st} 或 Tajima D) 的置信区间以得到可信的范围。然后将真实的群体遗传数据统计结果与这些置信区间相比较。

Biopython 溯祖模拟可以创建群体场景 (demographic scenarios) 和基因组，然后运行 SIMCOAL2。

12.2.1 创建场景 (scenario)

创建场景包括创建群体及其染色体结构。多数情况下（如计算近似贝斯估计量 (Approximate Bayesian Computations – ABC)），测试不同参数的变化很重要（如不同的有效群体大小 N_e ，从 10，50，500 到 1000 个体）。提供的代码可以很容易地模拟具有不同群体参数的场景。

下面我们将学习怎样创建场景，然后是怎样进行模拟。

12.2.1.1 群体

有一些内置的预定义群体，均包含两个共同的参数：同群种 (deme) 的样本大小（在模板中称为 `sample_size`，其使用请见下文）和同群种大小，如亚群大小 (`pop_size`)。所有的群体都可以作为模板，所有的参数也可以变化，每个模板都有自己的系统名称。这些预定义的群体/模板 (template) 包括：

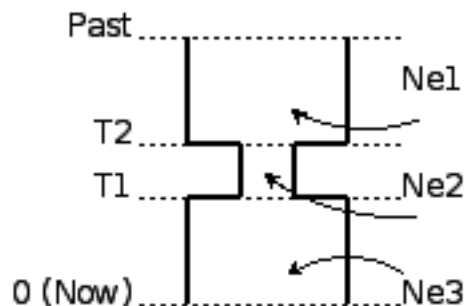
Single population, constant size 单一种群固定群体大小。标准的参数即可满足它，模板名称：`simple`。

Single population, bottleneck 单一种群瓶颈效应，如图 12.2.1.1 所示。参数有当前种群大小（图中 `ne3` 模板的 `pop_size`）、种群扩张时间 - 扩张发生后代数 (`expand_gen`)，瓶颈发生时的有效群体大小 (`ne2`)，种群收缩时间 (`contract_gen`) 以及原始种群大小 (`ne3`)。模板名：`bottle`。

Island model 典型的岛屿模型。同群种 (deme) 总数表示为 `total_demes`，迁移率表示为 `mig`。模板名：`island`。

Stepping stone model - 1 dimension 一维脚踏石模型 (Stepping stone model)，极端状态下种群分布不连续。同群种 (deme) 总数表示为 `total_demes`，迁移率表示为 `mig`。模板名：`ssm_1d`。

Stepping stone model - 2 dimensions 二维脚踏石模型，极端状态下种群分布不连续。参数有表示水平维度的 `x` 和表示垂直维度的 `y`（同群种总数即为 `x`），以及表示迁移率的 `mig`。模板名：`ssm_2d`。



在我们的第一个示例中，将生成一个单一种群固定群体大小 (Single population, constant size) 模板，样本大小 (sample size) 为 30，同群种大小 (deme size) 为 500。代码如下：

```
from Bio.PopGen.SimCoal.Template import generate_simcoal_from_template

generate_simcoal_from_template('simple',
    [(1, [('SNP', [24, 0.0005, 0.0])])])
```

```
[('sample_size', [30]),
 ('pop_size', [100])]
```

执行该段代码将会在当前目录生成一个名为 `simple_100_300.par` 的文件，该文件可作为 SIMCOAL2 的输入文件，用于模拟群体（下面将会展示 Biopython 是如何调用 SIMCOAL2）。

这段代码仅包含一个函数的调用，让我们一个参数一个参数地讨论。

第一个参数是模板 `id`（从上面的模板列表中选择）。我们使用 `'simple'`，表示的是单一群体固定种群大小模板。

第二个参数是染色体结构，将在下一节详细阐述。

第三个参数是所有需要的参数列表及其所有可能的值（此列中所有的参数都只含有一个可能值）。

现在让我们看看岛屿模型示例。我们希望生成几个岛屿模型，并对不同大小的同群种感兴趣：10、50 和 100，迁移率为 1%。样本大小和同群种大小与上一个示例一致，代码如下：

```
from Bio.PopGen.SimCoal.Template import generate_simcoal_from_template

generate_simcoal_from_template('island',
                               [(1, [('SNP', [24, 0.0005, 0.0])])],
                               [('sample_size', [30]),
                                ('pop_size', [100]),
                                ('mig', [0.01]),
                                ('total_demes', [10, 50, 100])])
```

此例将会生成 3 个文件：`island_100_0.01_100_30.par`，`island_10_0.01_100_30.par` 和 `island_50_0.01_100_30.par`。注意，生成文件名的规律是：模板名，然后是参数值逆序排列。

还有一些存在较多争议的群体模板（请见 Biopython 源代码中 `Bio/PopGen/SimCoal/data` 文件夹）。同时，用户可以创建新的模板，该功能将在以后的文档中讨论。

12.2.1.2 染色体结构

我们强烈建议你阅读 SIMCOAL2 文档，以完整理解染色体结构建模的各种使用。在本小节，我们只讨论如何使用 Biopython 接口实现指定的染色体结构，不会涉及 SIMCOAL2 可实现哪些染色体结构。

我们首先实现一条染色体，包含 24 个 SNPs，每个相邻基因座的重组率为 0.0005，次等位基因的最小频率为 0。这些由以下列表指定（作为第二个参数传递给 `generate_simcoal_from_template` 函数）：

```
[(1, [('SNP', [24, 0.0005, 0.0])])]
```

这实际上是上一个示例使用的染色体结构。

染色体结构表示为一个包含所有染色体的列表，每条染色体（即列表中的每个元素）由一个元组 (tuple) 组成，元组包括一对元素组成。元组的第一个元素是染色体被重复的次数（因为有可能需要多次重复同一条染色体）。元组的第二个元素是一个表示该染色体的实际组成的列表，每个列表元素又包括一对元素，第一个是基因座类型，第二个是该基因座的参数列表。是否有点混淆了呢？在我们展示示例之前，先让我们回顾下上一个示例：我们有一个列表（表示一条染色体），该染色体只有一个实例（因此不会被重复），它由 24 个 SNPs 组成，每个相邻 SNP 间的重组率为 0.0005，次等位基因的最小频率为 0.0（即它可以在某些染色体中缺失）。

现在让我们看看更复杂的示例：

```
[
  (5, [
    ('SNP', [24, 0.0005, 0.0])
  ]),
]
```

```
(2, [
    ('DNA', [10, 0.0, 0.00005, 0.33]),
    ('RFLP', [1, 0.0, 0.0001]),
    ('MICROSAT', [1, 0.0, 0.001, 0.0, 0.0])
])
]
```

首先，我们有 5 条与上一示例具有相同结构组成的染色体（即 24SNPs）。然后是 2 条这样的染色体：包含一段具有重组率为 0.0、突变率为 0.0005 及置换率为 0.33 的 10 个核苷酸长度的 DNA 序列，一段具有重组率为 0.0、突变率为 0.0001 的 RFLP，一段具有重组率为 0.0、突变率为 0.001、几何参数为 0.0、范围限制参数为 0.0 的微卫星（microsatellite, STR）序列（注意，因为这是单个微卫星，接下来没有基因座，因此这里的重组率没有任何影响，更多关于这些参数的信息请查阅 SIMCOAL2 文档，你可以使用它们模拟各种突变模型，包括典型的微卫星渐变突变模型）。

12.2.2 运行 SIMCOAL2

现在我们讨论如何从 Biopython 内部运行 SIMCOAL2。这需要 SIMCOAL2 的可执行二进制文件名为 simcoal2（在 Windows 平台下为 simcoal2.exe），请注意，从官网下载的程序命名格式通常为 simcoal2_x.y。因此，当安装 SIMCOAL2 时，需要重命名可执行文件，这样 Biopython 才能正确调用。

SIMCOAL2 可以处理不是使用上诉方法生成的文件（如手动配置的参数文件），但是我们将使用上述方法得到的文件创建模型：

```
from Bio.PopGen.SimCoal.Template import generate_simcoal_from_template
from Bio.PopGen.SimCoal.Controller import SimCoalController

generate_simcoal_from_template('simple',
[
    (5, [
        ('SNP', [24, 0.0005, 0.0])
    ]),
    (2, [
        ('DNA', [10, 0.0, 0.00005, 0.33]),
        ('RFLP', [1, 0.0, 0.0001]),
        ('MICROSAT', [1, 0.0, 0.001, 0.0, 0.0])
    ])
],
[('sample_size', [30]),
 ('pop_size', [100])])

ctrl = SimCoalController('.')
ctrl.run_simcoal('simple_100_30.par', 50)
```

需要注意的是最后两行（以及新增的 import 行）。首先是创建一个应用程序控制器对象，需要指定二进制可执行文件所在路径。

模拟器在最后一行运行：从上述阐述的规律可知，文件名为 simple_100_30.par 的输入文件是我们创建的模拟参数文件，然后我们指定了希望运行 50 次独立模拟。默认情况下，Biopython 模拟二倍体数据，但是可以添加第三个参数用于模拟单倍体数据（字符串 '0'）。然后，SIMCOAL2 将会执行（这需要运行很长时间），并创建一个包含模拟结果的文件夹，结果文件便可用于分析（尤其是研究 Arlequin3 数据）。在未来的 Biopython 版本中，可能会支持 Arlequin3 格式文件的读取，从而在 Biopython 中便能分析 SIMCOAL2 结果。

12.3 其它应用程序

这里我们讨论一些处理其它的群体遗传学中应用程序的接口和小工具，这些应用程序具有争议，使用得较少。

12.3.1 FDist：检测选择压力和分子适应

FDist 是一个选择压力检测的应用程序包，基于通过 F_{st} 和杂合度计算（即模拟）得到的“中性”(“neutral”) 置信区间。“中性”置信区间外的 Markers（可以是 SNPs，微卫星，AFLPs 等等）可以被认为是候选的受选择 marker。

FDist 主要运用在当 marker 数量足够用于估计平均 F_{st} ，而不足以从数据集中计算出离群点 - 直接地或者在知道大多数 marker 在基因组中的相对位置的情况下使用基于如 Extended Haplotype Heterozygosity (EHH) 的方法。

典型的 FDist 的使用如下：

1. 从其它格式读取数据为 FDist 格式；
2. 计算平均 F_{st} ，由 FDist 的 datacal 完成；
3. 根据平均 F_{st} 和期望的总群体数模拟“中性”markers，这是核心部分，由 FDist 的 fdist 完成；
4. 根据指定的置信范围（通常是 95% 或者是 99%）计算置信区间，由 cplot 完成，主要用于对区间作图；
5. 用模拟的“中性”置信区间评估每个 Marker 的状态，由 pv 完成，用于检测每个 marker 与模拟的相比的离群状态；

我们将以示例代码讨论每一步（FDist 可执行二进制文件需要在 PATH 环境变量中）。

FDist 数据格式是该应用程序特有的，不被其它应用程序使用。因此你需要转化你的数据格式到 FDist 可使用的格式。Biopython 可以帮助你完成这个过程。这里有一个将 GenePop 格式转换为 FDist 格式的示例（同时包括后面示例将用到的 import 语句）：

```
from Bio.PopGen import GenePop
from Bio.PopGen import FDist
from Bio.PopGen.FDist import Controller
from Bio.PopGen.FDist.Uutils import convert_genepop_to_fdist

gp_rec = GenePop.read(open("example.gen"))
fd_rec = convert_genepop_to_fdist(gp_rec)
in_file = open("infile", "w")
in_file.write(str(fd_rec))
in_file.close()
```

在该段代码中，我们解析 GenePop 文件并转化为 FDist 记录 (record)。

输出 FDist 记录将得到可以直接保存到可用于 FDist 的文件的字符串。FDist 需要输入文件名为 infile，因此我们将记录保存到文件名为 infile 的文件。

FDist 记录最重要的字段 (field) 是：num_pops，群体数量；num_loci，基因座数量和 loci_data，marker 数据。记录的许多信息对用户来说可能没有用处，仅用于传递给 FDist。

下一步是计算平均数据集的 F_{st} （以及样本大小）：

```
ctrl = Controller.FDistController()
fst, samp_size = ctrl.run_datacal()
```


第一行我们创建了一个控制调用 FDist 软件包的對象，該對象被用于调用該包的其它应用程序。

第二行我们调用 datacal 应用程序，它用于计算 F_{st} 和样本大小。值得注意的是，用 datacal 计算得到的 F_{st} 是 Weir-Cockerham θ 的“变种”(variation)。

现在我们可以调用主程序 fdist 模拟中性 Markers。

```
sim_fst = ctrl.run_fdist(npops = 15, nsamples = fd_rec.num_pops, fst = fst,
    sample_size = samp_size, mut = 0, num_sims = 40000)
```

npops 现存自然群体数量，完全是一个“瞎猜值”(“guestimate”)，必须小于 100。

nsamples 抽样群体数量，需要小于 npops。

fst 平均 F_{st} 。

sample_size 每个群体抽样个体平均数

mut 突变模型：0 - 无限等位基因突变模型；1 - 渐变突变模型

num_sims 执行模拟的次数。通常，40000 左右的数值即可，但是如果得到的执行区间范围比较大（可以通过下面的置信区间作图检测到），可以上调此值（建议每次调整 10000 次模拟）。

样本数量和样本大小措辞上的混乱源于原始的应用程序。

将会得到一个名为 out.dat 的文件，它包含模拟的杂合度和 F_{st} 值，行数与模拟的次数相同。

注意，fdist 返回它可以模拟的平均 F_{st} ，关于此问题更多的细节，请阅读下面的“估计期望的平均 F_{st} ”

下一步（可选步骤）是计算置信区间：

```
cpl_interval = ctrl.run_cplot(ci=0.99)
```

只能在运行 fdist 之后才能调用 cplot。

这将计算先前 fdist 结果的置信区间（此例中为 99%）。第一个元素是杂合度，第二个是该杂合度的 F_{st} 置信下限，第三个是 F_{st} 平均值，第四个是置信上限。可以用于记录置信区间等高线。该列表也可以输出到 out.cpl 文件。

这步的主要目的是返回一系列的点用于对置信区间作图。如果只是需要根据模拟结果对每个 marker 的状态进行评估，可以跳过此步。

```
pv_data = ctrl.run_pv()
```

只能在运行 fdist 之后才能调用 pv。

这将使用模拟 marker 对每个个体真实的 marker 进行评估，并返回一个列表，顺序与 FDist 记录中 loci_list 一致（loci_list 又与 GenePop 顺序一致）。列表中每个元素包含四个元素，其中最重要的是最后一个元素（关于其他的元素，为了简单起见，我们不在这里讨论，请见 pv 帮助文档），它返回模拟的 F_{st} 低于 marker F_{st} 的概率。较大值说明极有可能是正选择（positive selection）marker，较小值表明可能是平衡选择（balancing selection）marker，中间值则可能是中性 marker。怎样的值是“较大值”、“较小值”或者“中间值”是一个很主观的问题，但当使用置信区间方法及 95% 的置信区间时，“较大值”在 0.95 - 1.00 之间，“较小值”在 0.00 - 0.05 之间，“中间值”在 0.05 - 0.05 之间。

12.3.1.1 估计期望的平均 F_{st}

FDist 通过对由下例公式得到的迁移率进行溯祖模拟估计期望的平均 F_{st} ：

$$N_m = \frac{1 - F_{st}}{4F_{st}}$$

该公式有一些前提，比如种群大小无限大。

在实践中，当群体数量比较小，突变模型为渐进突变模型，样本大小增加，`fdist` 将不能模拟得到可接受的近似平均 F_{st} 。

为了解决这个问题，Biopython 提供了一个使用迭代方法的函数，通过依次运行几个 `fdist` 得到期望的值。该方法比运行单个 `fdist` 相比耗费更多计算资源，但是可以得到更好的结果。以下代码运行 `fdist` 得到期望的 F_{st} ：

```
sim_fst = ctrl.run_fdist_force_fst(npops = 15, nsamples = fd_rec.num_pops,
    fst = fst, sample_size = samp_size, mut = 0, num_sims = 40000,
    limit = 0.05)
```

与 `run_fdist` 相比，唯一一个新的可选参数是 `limit`，它表示期望的最大错误率。`run_fdist` 可以（或许应该）由 `run_fdist_force_fst` 替代。

12.3.1.2 说明

计算平均 F_{st} 的过程可能比这里呈现的要复杂得多。更多的信息请查阅 `FDist` README 文件。同时，Biopython 的代码也可用于实现更复杂的过程。

12.4 未来发展

最期望的是您的参与！

尽管如此，已经完成的功能模块正在逐步加入到 `Bio.PopGen`，这些代码覆盖了程序 `FDist` 和 `SimCoal2`，`HapMap` 和 `UCSC Table Browser` 数据库，以及一些简单的统计计算，如 F_{st} ，或等位基因数。

第 13 章 BIO.PHYLO 系统发育分析

Biopython1.54 开始引入了 Bio.Phylo 模块，与 SeqIO 和 AlignIO 类似，它的目的是提供一个通用的独立于源数据格式的方法来使用系统进化树，同时提供一致的 API 来进行 I/O 操作。

Bio.Phylo 在一篇开放获取的期刊文章中有介绍 [9, Talevich *et al.*, 2012], 这可能对您也有所帮助。

13.1 示例：树中有什么？

为了熟悉这个模块，让我们首先从一个已经创建好的树开始，从几个不同的角度来审视它。接着我们将给树的分支上颜色，并使用一个特殊的 phyloXML 特性，最终保存树。

译者注：本翻译中，分支对应原文中的 **branch**，原文中一般代表某一个节点的前导连线；而进化枝对应原文中的 **clade**，代表某个节点所代表的整个进化分支，包括本身和它所有的后代；若 clade 代表 biopython 中的对象则保留原文

在终端中使用你喜欢的编辑器创建一个简单的 Newick 文件：

```
% cat > simple.dnd <<EOF
> (((A,B),(C,D)),(E,F,G));
> EOF
```

这棵树没有分支长度，只有一个拓扑结构和标记的端点。（如果你有一个真正的树文件，你也可以使用它来替代进行示例操作。）

选择启动你的 Python 解释器：

```
% ipython -pylab
```

对于交互式操作，使用参数 -pylab 启动 IPython 解释器能启用 matplotlib 整合功能，这样图像就能自动弹出来。我们将在这个示例中使用该功能。

现在，在 Python 终端中，读取树文件，给定文件名和格式名。

```
>>> from Bio import Phylo
>>> tree = Phylo.read("simple.dnd", "newick")
```

以字符串打印该树对象我们将得到整个对象的层次结构概况。

```
>>> print tree

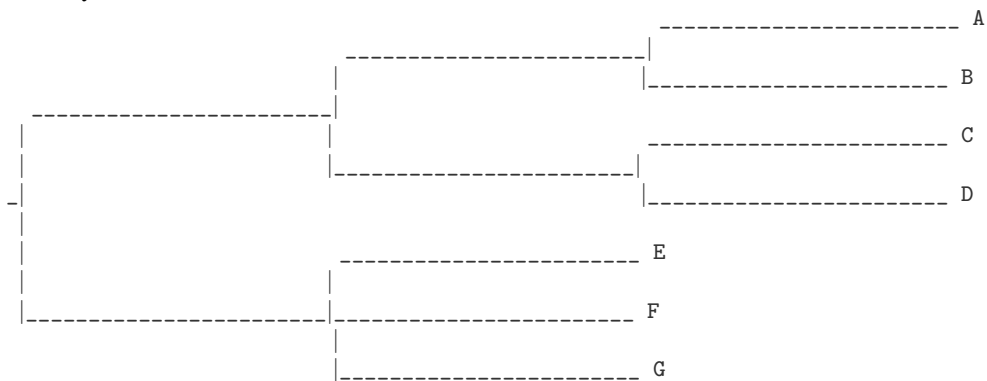
Tree(weight=1.0, rooted=False, name="")
  Clade(branch_length=1.0)
    Clade(branch_length=1.0)
      Clade(branch_length=1.0)
        Clade(branch_length=1.0, name="A")
```

```
        Clade(branch_length=1.0, name="B")
    Clade(branch_length=1.0)
        Clade(branch_length=1.0, name="C")
        Clade(branch_length=1.0, name="D")
    Clade(branch_length=1.0)
        Clade(branch_length=1.0, name="E")
        Clade(branch_length=1.0, name="F")
        Clade(branch_length=1.0, name="G")
```

Tree 对象包含树的全局信息，如树是有根树还是无根树。它包含一个根进化枝，和以此往下以列表嵌套的所有进化枝，直至叶子分支。

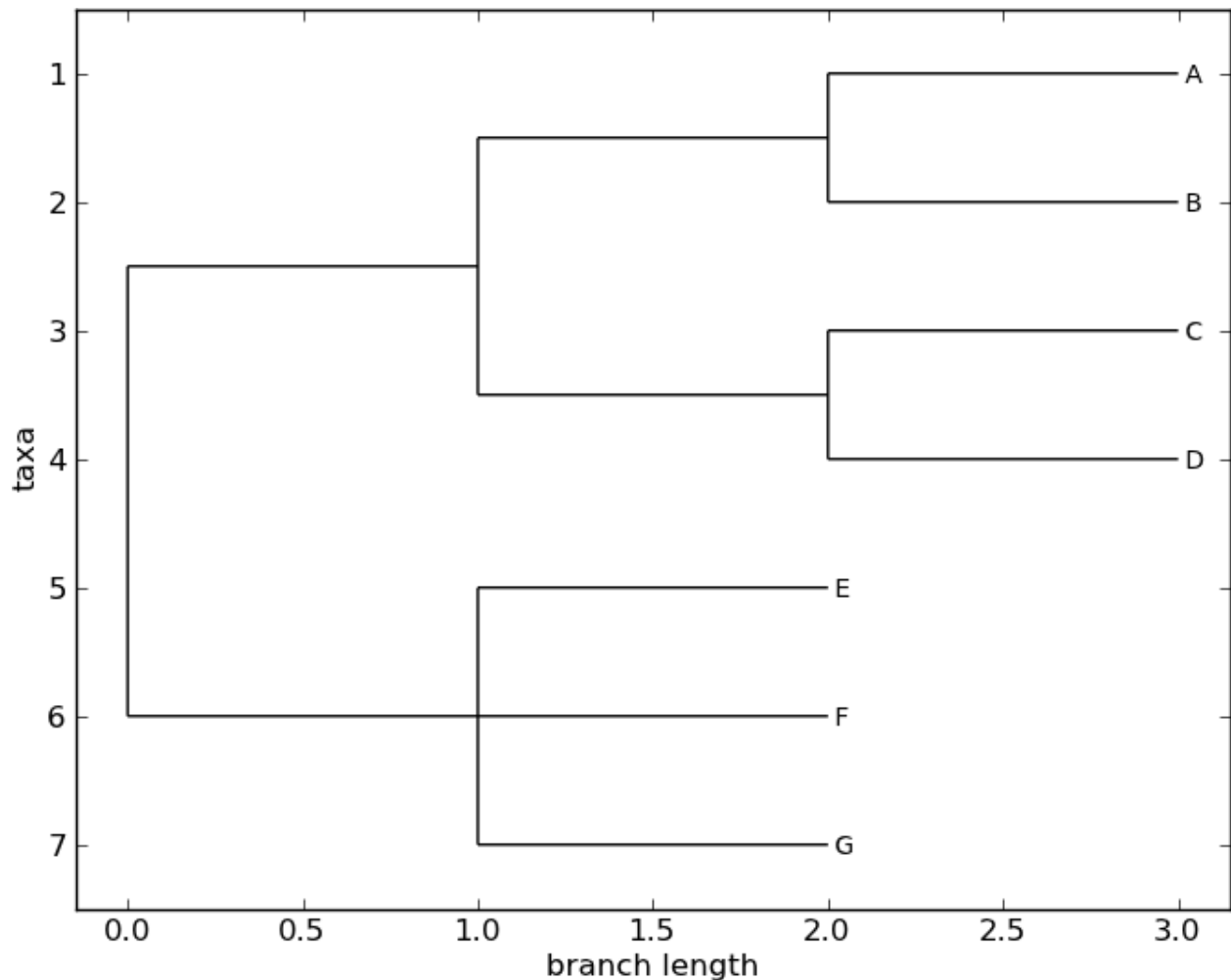
函数 `draw_ascii` 创建一个简单的 ASCII-art(纯文本) 系统发生图。在没有更好图形工具的情况下，这对于交互研究来说是一个方便的可视化展示方式。

```
>>> Phylo.draw_ascii(tree)
```



如果你安装有 `matplotlib` 或者 `pylab`, 你可以使用 `draw` 函数一个图像 (见 Fig. 13.1):

```
>>> tree.rooted = True
>>> Phylo.draw(tree)
```



13.1.1 给树的分支上颜色

函数 `draw` 和 `draw_graphviz` 支持在树中显示不同的颜色和分支宽度。从 Biopython 1.59 开始，Clade 对象就开始支持 `color` 和 `width` 属性，且使用他们不需要额外支持。这两个属性都表示导向给定的进化枝前面的分支的属性，并依次往下作用，所以所有的后代分支在显示时也都继承相同的宽度和颜色。

在早期的 Biopython 版本中，PhyloXML 树有些特殊的特性，使用这些属性需要首先将这个树转换为一个基本树对象的子类 `Phylogeny`，该类在 `Bio.Phylo.PhyloXML` 模块中。

在 Biopython 1.55 和之后的版本中，这是一个很方便的树方法：

```
>>> tree = tree.as_phyloxml()
```

在 Biopython 1.54 中，你能通过导入一个额外的模块实现相同的事情：

```
>>> from Bio.Phylo.PhyloXML import Phylogeny
>>> tree = Phylogeny.from_tree(tree)
```

注意 Newick 和 Nexus 文件类型并不支持分支颜色和宽度，如果你在 `Bio.Phylo` 中使用这些属性，你只能保存这些值到 PhyloXML 格式中。（你也可以保存成 Newick 或 Nexus 格式，但是颜色和宽度信息在输出的文件时会被忽略掉。）

现在我们开始指定颜色。首先，我们将设置根进化枝为灰色。我们能够通过赋值 24 位的颜色值来实现，用三位数的 RGB 值、HTML 格式的十六进制字符串、或者预先设置好的颜色名称。

```
>>> tree.root.color = (128, 128, 128)
```

Or:

```
>>> tree.root.color = "#808080"
```

Or:

```
>>> tree.root.color = "gray"
```

一个进化枝的颜色会被当作从上而下整个进化枝的颜色，所以我们这里设置根的颜色会将整个树的颜色变为灰色。我们能够通过为树中下面分支赋值不同的颜色来重新定义某个分支的颜色。

让我们先定位“E”和“F”最近祖先（MRCA）节点。方法 `common_ancestor` 返回原始树中这个进化枝的引用，所以当我们设置该进化枝为“salmon”颜色时，这个颜色则会在原始的树中显示出来。

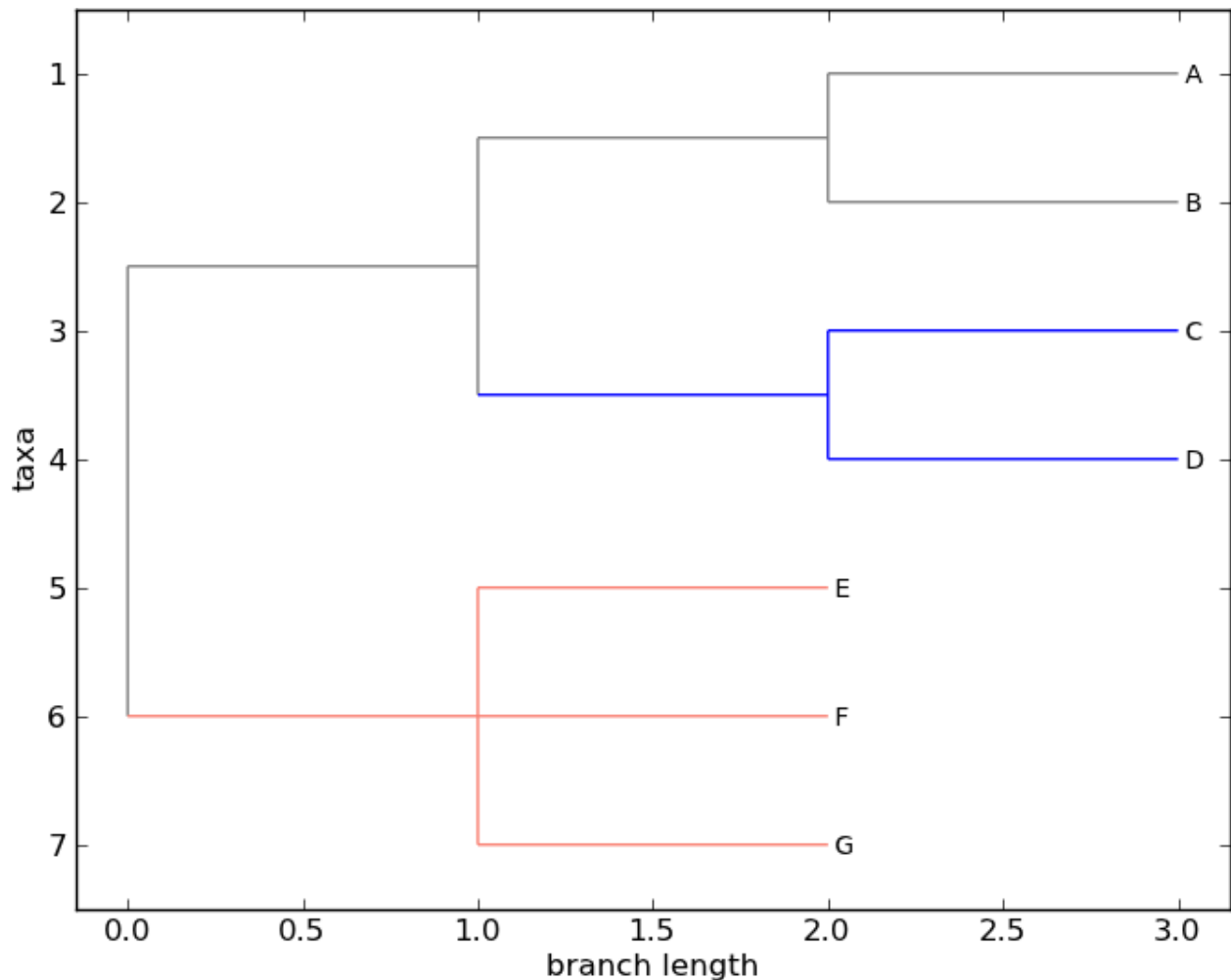
```
>>> mrca = tree.common_ancestor({"name": "E"}, {"name": "F"})
>>> mrca.color = "salmon"
```

当我们碰巧明确地知道某个进化枝在树中的位置，以嵌套列表的形式，我们就能通过索引的方式直接跳到那个位置。这里，索引 `[0,1]` 表示根节点的第一个子代节点的第二个子代。

```
>>> tree.clade[0,1].color = "blue"
```

最后，展示一下我们的工作结果（见 Fig. 13.1.1）:

```
>>> Phylo.draw(tree)
```



注意进化枝的颜色包括导向它的分支和它的子代的分支。E 和 F 的共同祖先结果刚好在根分支下面，而通过这样上色，我们能清楚的看出这个树的根在哪里。

我们已经完成了很多！现在让我们休息一下，保存一下我们的工作。使用一个文件名或句柄（这里我们使用标准输出来查看将会输出什么）和 phyloxml 格式来调用 write 函数。PhyloXML 格式保存了我们设置的颜色，所以你能通过其他树查看工具，如 Archaeopteryx，打开这个 phyloXML 文件，这些颜色也会显示出来。

```
>>> import sys
>>> Phylo.write(tree, sys.stdout, "phyloxml")

<phy:phyloxml xmlns:phy="http://www.phyloxml.org">
  <phy:phylogeny rooted="true">
    <phy:clade>
      <phy:branch_length>1.0</phy:branch_length>
      <phy:color>
        <phy:red>128</phy:red>
        <phy:green>128</phy:green>
        <phy:blue>128</phy:blue>
      </phy:color>
    </phy:clade>
    <phy:branch_length>1.0</phy:branch_length>
    <phy:clade>
      <phy:branch_length>1.0</phy:branch_length>
```



```
<phy:clade>
  <phy:name>A</phy:name>
  ...
```

本章的其余部分将更加细致的介绍 Bio.Phylo 核心功能。关于 Bio.Phylo 的更多例子，请参见 Biopython.org 上的 Cookbook 手册页面。

http://biopython.org/wiki/Phylo_cookbook

13.2 I/O 函数

和 SeqIO、AlignIO 类似, Phylo 使用四个函数处理文件的输入输出: `parse`、`read`、`write` 和 `convert`，所有的函数都支持 Newick、NEXUS、phyloXML 和 NeXML 等树文件格式。

`read` 函数解析并返回给定文件中的单个树。注意，如果文件中包含多个或不包含任何树，它将抛出一个错误。

```
>>> from Bio import Phylo
>>> tree = Phylo.read("Tests/Nexus/int_node_labels.nwk", "newick")
>>> print tree
```

(Biopython 发布包的 Tests/Nexus/ 和 Tests/PhyloXML/ 文件夹中有相应的例子)

处理多个 (或者未知个数) 的树文件，需要使用 `parse` 函数迭代给定文件中的每一个树。

```
>>> trees = Phylo.parse("Tests/PhyloXML/phyloxml_examples.xml", "phyloxml")
>>> for tree in trees:
...     print tree
```

使用 `write` 函数输出一个或多个可迭代的树。

```
>>> trees = list(Phylo.parse("phyloxml_examples.xml", "phyloxml"))
>>> tree1 = trees[0]
>>> others = trees[1:]
>>> Phylo.write(tree1, "tree1.xml", "phyloxml")
1
>>> Phylo.write(others, "other_trees.xml", "phyloxml")
12
```

使用 `convert` 函数转换任何支持的树格式。

```
>>> Phylo.convert("tree1.dnd", "newick", "tree1.xml", "nexml")
1
>>> Phylo.convert("other_trees.xml", "phyloxml", "other_trees.nex", "nexus")
12
```

和 SeqIO 和 AlignIO 类似，当使用字符串而不是文件作为输入输出时，需要使用“`StringIO`”函数。

```
>>> from Bio import Phylo
>>> from StringIO import StringIO
>>> handle = StringIO("((A,B),(C,D)),(E,F,G));")
>>> tree = Phylo.read(handle, "newick")
```

13.3 查看和导出树


了解一个 Tree 对象概况的最简单的方法是用 `print` 函数将它打印出来：

```
>>> tree = Phylo.read("Tests/PhyloXML/example.xml", "phyloxml")
>>> print tree
Phylogeny(rooted='True', description='phyloXML allows to use either a "branch_length"
attribute...', name='example from Prof. Joe Felsenstein's book "Inferring Phyl...')
  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')
```

上面实际上是 Biopython 的树对象层次结构的一个概况。然而更可能的情况是，你希望见到画出树的形状，这里三个函数来做这件事情。

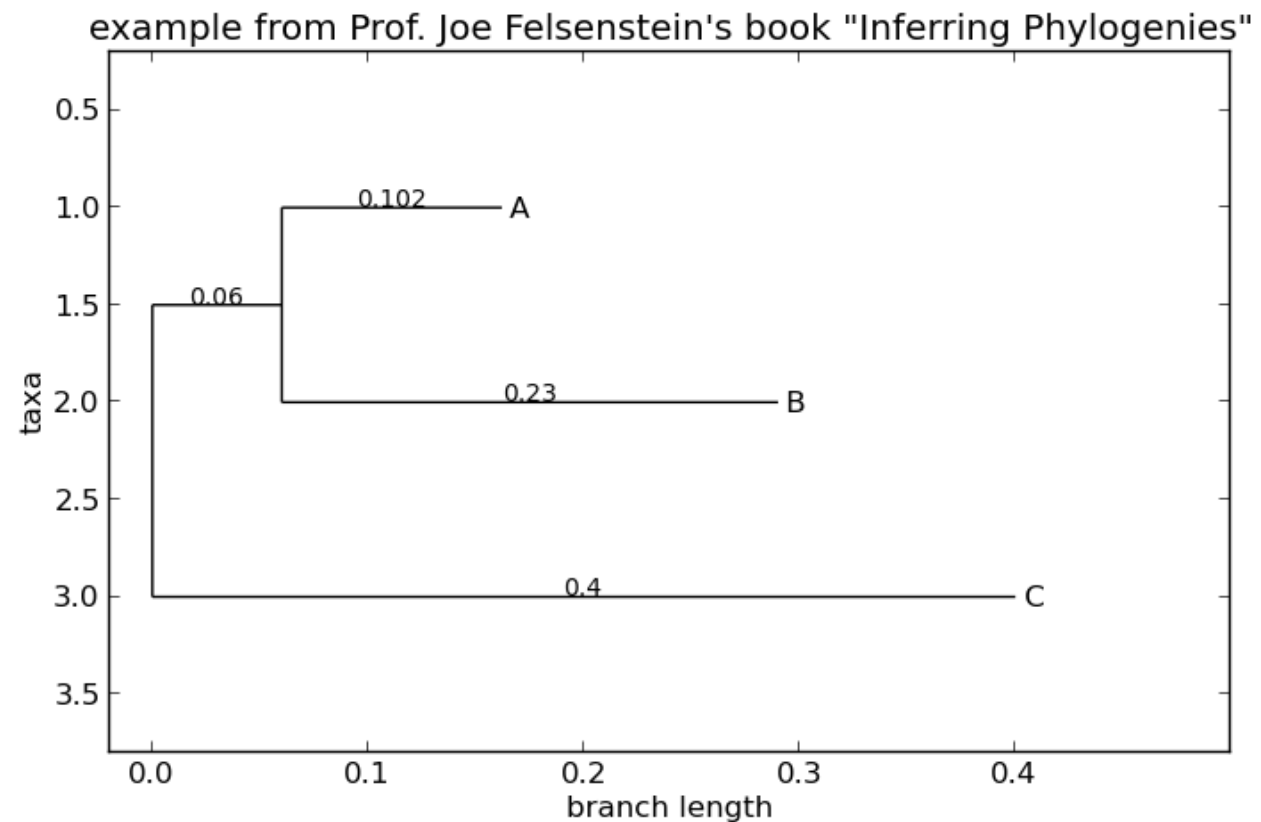
如我们在 demo 中看到的一样，`draw_ascii` 打印一个树的 ascii-art 图像（有根进化树）到标准输出，或者一个打开的文件句柄，若有提供。不是所有关于树的信息被显示出来，但是它提供了一个不依赖于任何外部依赖的快速查看树的方法。

```
>>> tree = Phylo.read("example.xml", "phyloxml")
>>> Phylo.draw_ascii(tree)
```



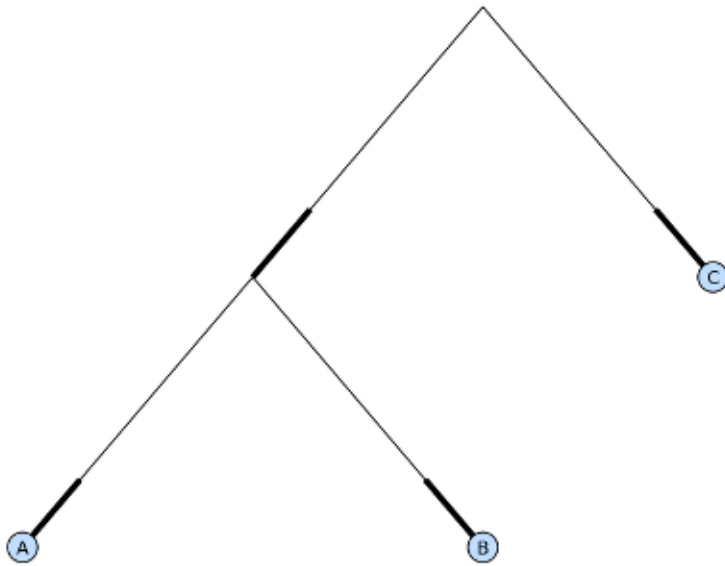
`draw` 函数则使用 `matplotlib` 类库画出一个更加好看的图像。查看 API 文档以获得关于它所接受的用来定制输出的参数。

```
>>> tree = Phylo.read("example.xml", "phyloxml")
>>> Phylo.draw(tree, branch_labels=lambda c: c.branch_length)
```



`draw_graphviz` 则画出一个无根的进化分枝图 (cladogram)，但是它要求你安装有 Graphviz、PyDot 或 PyGraphviz、Network 和 matplotlib (或 pylab)。使用上面相同的例子，和 Graphviz 中的 `dot` 程序，让我们来画一个有根树 (见图. 13.3)：

```
>>> tree = Phylo.read("example.xml", "phyloxml")
>>> Phylo.draw_graphviz(tree, prog='dot')
>>> import pylab
>>> pylab.show() # Displays the tree in an interactive viewer
>>> pylab.savefig('phylo-dot.png') # Creates a PNG file of the same graphic
```

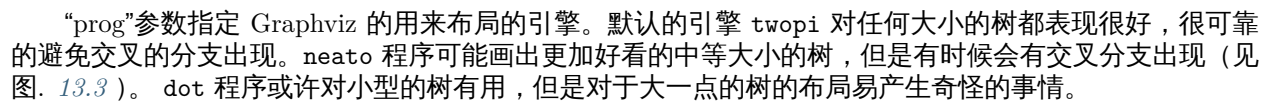


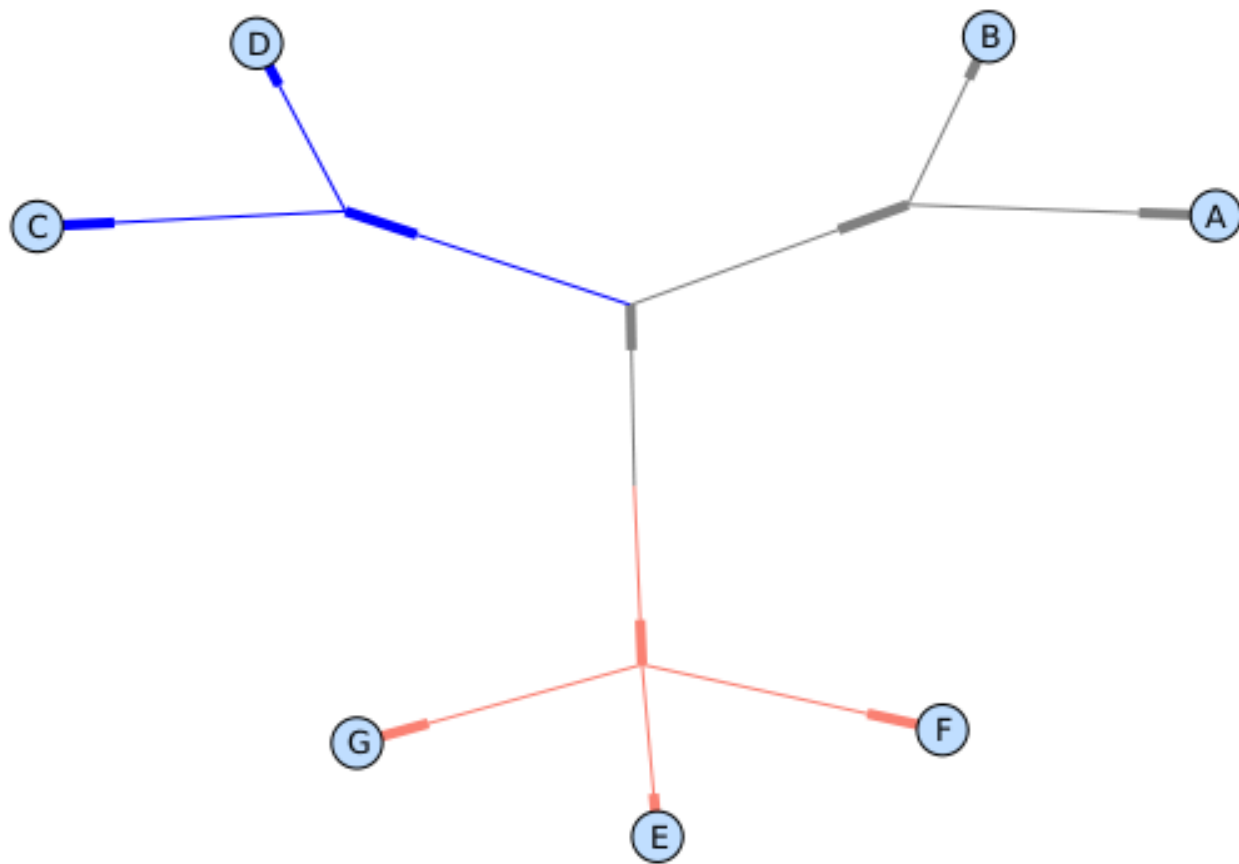
(提示：如果你使用 `-pylab` 选项执行 IPython，调用 `draw_graphviz` 将导致 matplotlib 查看器自动运行，而不需要手动的调用 `show()` 方法。)

这将输出树对象到一个 NetworkX 图中，使用 Graphviz 来布局节点的位置，并使用 matplotlib 来显示它。这里有几个关键词参数来修改结果图像，包括大多数被 NetworkX 函数 `networkx.draw` 和 `networkx.draw_graphviz` 所接受的参数。

最终的显示也受所提供的树对象的 `rooted` 属性的影响。有根树在每个分支 (branch) 上显示一个“head”来表明它的方向 (见图. 13.3)：

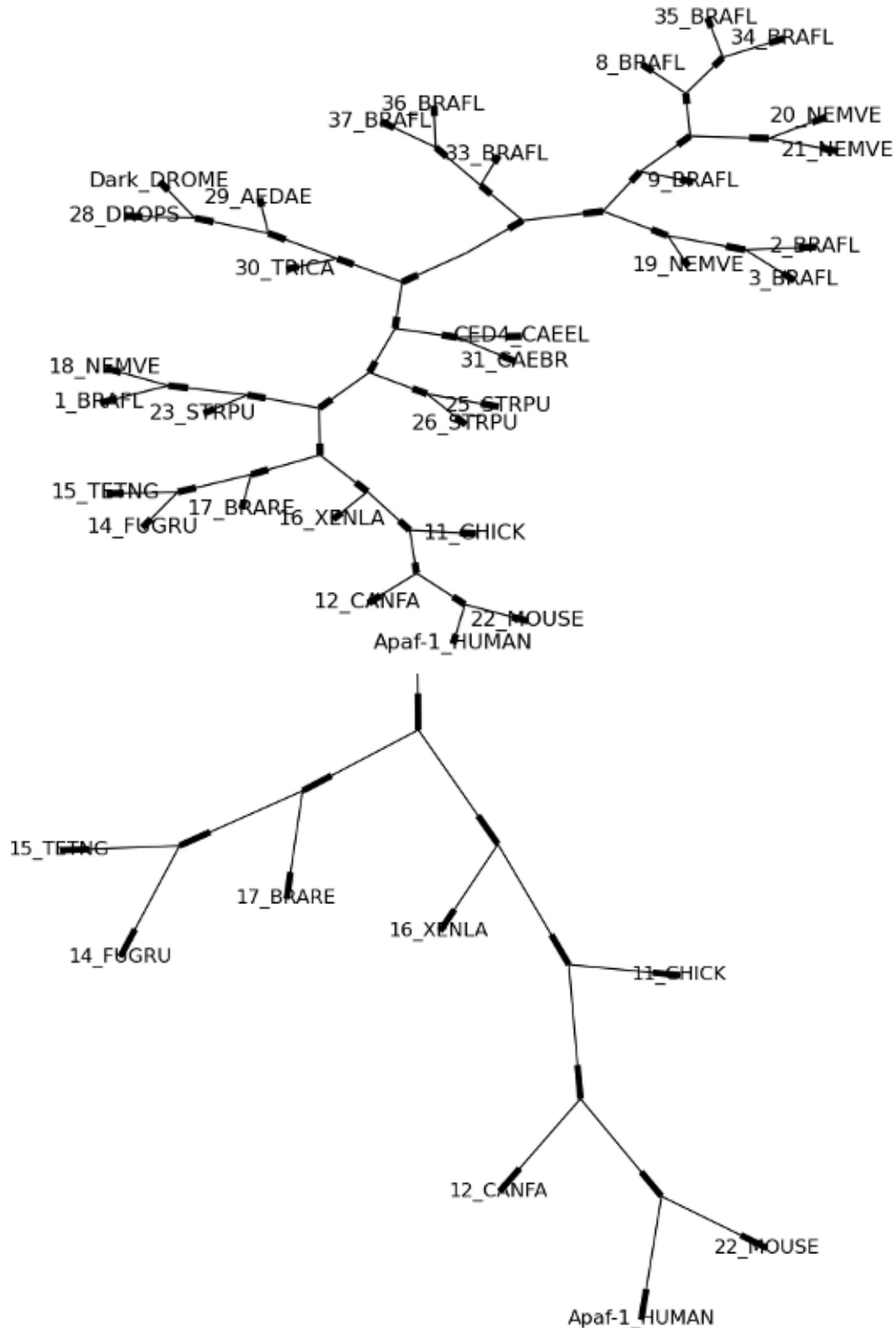
```
>>> tree = Phylo.read("simple.dnd", "newick")
>>> tree.rooted = True
>>> Phylo.draw_graphviz(tree)
```





这个查看方式非常方便研究大型的树，因为 matplotlib 查看器可以放大选择的区域，使得杂乱的图像变得稀疏。

```
>>> tree = Phylo.read("apaf.xml", "phyloxml")
>>> Phylo.draw_graphviz(tree, prog="neato", node_size=0)
```



注意，分支长度并没有被正确地显示，因为 Graphviz 在布局时忽略了他们。然而，分支长度可以在输出树为 NetworkX 图对象（`to_networkx`）时重新获得。

查看 Biopython 维基的 Phylo 页面 (<http://biopython.org/wiki/Phylo>) 以获得关于 `draw_ascii`、`draw_graphviz` 和 `to_networkx` 的更加高级的功能的描述和例子。

13.4 使用 Tree 和 Clade 对象

`parse` 和 `read` 方法产生的 `Tree` 对象是一些包含递归的子树的容器，连接到 `Tree` 对象的 `root` 属性（不管进化树实际上被认为是是否有根）。一个 `Tree` 包含进化树的全局信息，如有根性（rootedness）和指向一个单独的 `Clade` 的引用；一个 `Clade` 包含节点和进化枝特异性信息，如分支长度（branch length）和一个它自身后代 `Clade` 实例的列表，附着在 `clades` 属性上。

所以，这里 `tree` 和 `tree.root` 间是有区别的。然而，实际操作中，你几乎不需要担心它。为了缓和这个不同，`Tree` 和 `Clade` 两者都继承自 `TreeMixin`，它包含常用的用来查找、审视和修改树和任何它的进化枝的方法的实现。这意味着，所有 `tree` 所支持的方法在 `tree.root` 和任何它下面的 `clade` 中都能用。（`Clade` 也有一个 `root` 属性，它返回 `clade` 对象本身。）

13.4.1 查找和遍历类方法

为了方便起见，我们提供了两个简化的方法来直接返回所有的外部或内部节点为列表：

“`get_terminals`” 创建一个包含树的所有末端（叶子）节点的列表。

“`get_nonterminals`” 创建一个包含树的所有非末端（内部）节点的列表。

这两个都包装了一个能完全控制树的遍历的方法 `find_clades`。另外两个遍历方法 `find_elements` 和 `find_any` 依赖于同样的核心功能，也接受同样的参数，没有更好的描述我们就把这个参数叫做“目标说明”(target specification) 吧。它们指定哪些树中的对象将被匹配并在迭代过程中返回。第一个参数可以是下面的任何类型：

- 一个 `TreeElement` 实例，那个树的元素将根据一致性被匹配——这样，使用 `Clade` 实例作为目标将找到树中的这个 `Clade`；
- 一个 `string`，匹配树元素的字符串表示——特别地，`Clade` 的 `name`（在 *Biopython 1.56* 中引入）；
- 一个 `class` 或 `type`，这样每一个类型（或子类型）相同的树元素都被匹配；
- 一个 `dictionary`，其中键（key）是树元素的属性名，值（value）将匹配到每个树元素相应的属性值。它变得更加详细：
 - 如果提供的是 `int` 类型，它将匹配数值上相等的属性，即，1 将匹配 1 或者 1.0
 - 如果提供的是 `boolean` 类型（`True` 或者 `False`），对应的属性值将被当做 `boolean` 求值和检验
 - `None` 匹配 `None`
 - 如果提供的是字符串，将被当做正则表达式对待（必须匹配对应元素属性的全部，不能只是前面的部分）。提供没有特殊正则表达式字符的字符串将精准的匹配字符串属性，所以如果你不适用正则表达式，不用担心它。例如，包含进化枝名称 `Foo1`、`Foo2` 和 `Foo3` 的一个树，`tree.find_clades({"name": "Foo1"})` 将匹配 `Foo1`，`{"name": "Foo.*"}` 匹配所有的三个进化枝，而 `{"name": "Foo"}` 并不匹配任何进化枝。

由于浮点数值可能产生奇怪的行为，我们不支持直接匹配 `floats` 类型。作为替代，使用 `boolean` 值 `True` 来匹配每个元素中指定属性的非零值，然后再对这个属性用不等式（或精确地数值，如果你喜欢危险地活着）进行手动过滤。

如果该字典包含多个条目，匹配的元素必须匹配所有给定的属性值——以“and”方式思考，而不是“or”。

- 一个接受一个参数（它将应用于树中的每一个元素），返回 `True` 或 `False` 的函数 `function`。为方便起见，`LookupError`、`AttributeError` 和 `ValueError` 被沉默，这样就提供了另外一个在树中查找浮点值的安全方式，或者一些更加复杂的特性。

在目标参数后面，有两个可选的关键词参数：

terminal —用来选择或排除末端进化枝（或者叫叶子节点）的一个 boolean 值：True 仅搜索末端进化枝，False 则搜索非末端（内部）进化枝，而默认为 None，同时搜索末端和非末端进化枝，包括没有 `is_terminal` 方法的任何树元素。

order —树遍历的顺序：“preorder”（默认值）是深度优先搜索（depth-first search, DFS），“postorder”是子节点先于父节点的 DFS 搜索，“level”是宽度优先搜索（breadth-first search, BFS）。

最后，这些方法接受任意的关键词参数，这些参数将被以和词典“目标说明”相同的方式对待：键表示要搜索的元素属性的名称，参数值（string、integer、None 或者 boolean）将和找到的每个属性的值进行比较。如果没有提供关键词参数，则任何 `TreeElement` 类型将被匹配。这个的代码普遍比传入一个词典作为“目标说明”要短：`tree.find_clades({"name": "Foo1"})` 可以简化为 `tree.find_clades(name="Foo1")`。

（在 Biopython 1.56 和以后的版本中，这可以更短：`tree.find_clades("Foo1")`）

现在我们已经掌握了“目标说明”，这里有一些遍历树的方法：

“**find_clades**” 查找每个包含匹配元素的进化枝。就是说，用 `find_elements` 查找每个元素，然而返回对应的 `clade` 对象。（这通常是你想要的。）

最终的结果是一个包含所有匹配对象的迭代器，默认为深度优先搜索。这不一定是和 Newick、Nexus 或 XML 原文件中显示的相同的顺序。

“**find_elements**”

查找和给定属性匹配的所有树元素，返回匹配的元素本身。简单的 Newick 树没有复杂的子元素，所以它将与 `find_clades` 的行为一致。PhyloXML 树通常在 `clade` 上附加有复杂的对象，所以这个方法对提取这些信息非常有用。

“**find_any**” 返回 `find_elements()` 所找到的第一个元素，或者 None。这对于检测树中是否存在匹配的元素也非常有用，可以在条件判断语句中使用。

另外两个用于帮助在树的节点间导航的方法：

“**get_path**” 直接列出从树的根节点（或当前进化枝）到给定的目标间的所有 `clade`。返回包含这个路径上所有 `clade` 对象的列表，以给定目标为结尾，但不包含根进化枝。

“**trace**” 列出树中两个目标间的所有 `clade` 对象，不包含起始和结尾。

13.4.2 信息类方法

这些方法提供关于整个树（或任何进化枝）的信息。

“**common_ancestor**” 查找所提供的所有目标的最近共同祖先（the most recent common ancestor）（这将是一个 `Clade` 对象）。如果没有提供任何目标，将返回当前 `Clade`（调用该方法的那个）的根；如果提供一个目标，将返回目标本身。然而，如果有任何提供的目标无法在当前 `tree`（或 `clade`）中找到，将引起一个异常。

“**count_terminals**” 计算树中末端（叶子）节点的个数。

“**depths**” 创建一个树中进化枝到其深度的映射。结果是一个字典，其中键是树中所有的 `Clade` 实例，值是从根到每个 `clade`（包含末端）的距离。默认距离是到这个 `clade` 的分支长度累加，然而使用 `unit_branch_lengths=True` 选项，将只计算分支的个数（其在树中的级数）。

“**distance**” 计算两个目标间的分支长度总和。如果只指定一个目标，另一个则为该树的根。

“**total_branch_length**” 计算这个树中的分支长度总和。这在系统发生学中通常就称为树的长度“length”，但是我们使用更加明确的名称，以避免和 Python 的术语混淆。

余下的方法是 boolean 检测方法：

“**is_bifurcating**” 如果树是严格的二叉树；即，所有的节点有 2 个或者 0 个子代（对应的，内部或外部）。根节点可能有一个后代，然而仍然被认为是二叉树的一部分。

“**is_monophyletic**” 检验给定的所有目标是否组成一个完成的子进化枝——即，存在一个进化枝满足：它的末端节点和给定的目标是相同的集合。目标需要时树中的末端节点。为方便起见，若给定目标是一个单系 (monophyletic)，这个方法将返回它们的共同祖先 (MCRA) (而不是 True)，否则将返回 False。

“**is_parent_of**” 若目标是这个树的后代 (descendant) 则为 True——不必为直接后代。检验一个进化枝的直接后代，只需要用简单的列表成员检测方法：`if subclade in clade: ...`

“**is_preterminal**” 若所有的直接后代都为末端则为 True；否则任何一个直接后代不为末端则为 False。

13.4.3 修改类方法

这些方法都在原地对树进行修改，所以如果你想保持原来的树不变，你首先要使用 Python 的 `copy` 模块对树进行完整的拷贝：

```
tree = Phylo.read('example.xml', 'phyloxml')
import copy
newtree = copy.deepcopy(tree)
```

“**collapse**” 从树中删除目标，重新连接它的子代 (children) 到它的父亲节点 (parent)。

“**collapse_all**” 删除这个树的所有后代 (descendants)，只保留末端节点 (terminals)。分支长度被保留，即到每个末端节点的距离保持不变。如指定一个目标 (见上)，只坍塌 (collapses) 和指定匹配的内部节点。

“**ladderize**” 根据末端节点的个数，在原地对进化枝 (clades) 进行排序。越深的进化枝默认被放到最后，使用 `reverse=True` 将其放到最前。

“**prune**” 从树中修剪末端进化枝 (terminal clade)。如果分类名 (taxon) 来自一个二叉枝 (bifurcation)，连接的节点将被坍塌，它的分支长度将被加到剩下的末端节点上。这可能不再是一个有意义的值。

“**root_with_outgroup**” 使用包含给定目标的外群进化枝 (outgroup clade) 重新确定树的根节点，即外群的共同祖先。该方法只在 `Tree` 对象中能用，不能用于 `Clade` 对象。

如果外群和 `self.root` 一致，将不发生改变。如果外群进化枝是末端 (即一个末端节点被作为外群)，一个新的二叉根进化枝将被创建，且到给定外群的分支长度为 0。否则，外群根部的内部节点变为整个树的一个三叉根。如果原先的根是一个二叉，它将被从树中遗弃。

在所有的情况下，树的分支长度总和保持不变。

“**root_at_midpoint**” 重新选择树中两个最远的节点的中点作为树的根。(这实际上是使用 `root_with_outgroup` 函数。)

“**split**” 产生 n (默认为 2) 个新的后代。在一个物种树中，这是一个物种形成事件。新的进化枝拥有给定的 `branch_length` 以及和这个进化枝的根相同的名字，名字后面包含一个整数后缀 (从 0 开始计数)——例如，分割名为“A”的进化枝将生成子进化枝“A0”和“A1”。

查看 Biopython 维基的 `Phylo` 页面 (<http://biopython.org/wiki/Phylo>) 以获得更多已有方法的使用示例。

13.4.4 PhyloXML 树的特性

phyloXML 文件格式包含用来注释树的，采用额外数据格式和图像提示的字段。

参加 Biopython 维基上的 `PhyloXML` 页面 (<http://biopython.org/wiki/PhyloXML>) 以查看关于使用 `PhyloXML` 提供的额外注释特性的描述和例子。

13.5 运行外部程序

尽管 Bio.Phylo 本身不从序列比对推断进化树，但这里有一些第三方的程序可以使用。他们通过 Bio.Phylo.Applications 模块获得支持，使用和 Bio.Emboss.Applications、Bio.Align.Applications 以及其他模块相同的通用框架。

Biopython 1.58 引入了一个 PhyML 的打包程序 (wrapper) (<http://www.atgc-montpellier.fr/phyml/>)。该程序接受一个 phymlp-relaxed 格式 (它是 Phymlp 格式，然而没有对分类名称的 10 个字符的限制) 的比对输入和多种参数。一个快速的例子是：

```
>>> from Bio import Phylo
>>> from Bio.Phylo.Applications import PhymlCommandline
>>> cmd = PhymlCommandline(input='Tests/Phymlp/random.phy')
>>> out_log, err_log = cmd()
```

这生成一个树文件和一个统计文件，名称为：`[input filename].phyml_tree.txt` 和 `[input filename].phyml_stats.txt`。树文件的格式是 Newick 格式：

```
>>> tree = Phylo.read('Tests/Phymlp/random.phy_phyml_tree.txt', 'newick')
>>> Phylo.draw_ascii(tree)
```

一个类似的 RAXML 打包程序 (<http://sco.h-its.org/exelixis/software.html>) 也已经被添加到 Biopython 1.60 中。

注意，如果你系统中已经安装了 EMBOSS 的 Phymlp 扩展，一些常用的 Phymlp 程序，包括 `dnaml` 和 `protml` 已经通过 Bio.Emboss.Applications 中的 EMBOSS 打包程序被支持。参见章节 6.4 以查看使用这些程序的例子和提示。

13.6 PAML 整合

Biopython 1.58 引入了对 PAML 的支持 (<http://abacus.gene.ucl.ac.uk/software/paml.html>)，它是一个采用最大似然法 (maximum likelihood) 进行系统进化分析的程序包。目前，对程序 `codeml`、`baseml` 和 `yn00` 的支持已经实现。由于 PAML 使用控制文件而不是命令行参数来控制运行时选项，这个打包程序 (wrapper) 的使用格式和 Biopython 的其他应用打包程序有些差异。

一个典型的流程是：初始化一个 PAML 对象，指定一个比对文件，一个树文件，一个输出文件和工作路径。下一步，运行时选项通过 `set_options()` 方法或者读入一个已有的控制文件来设定。最后，程序通过 `run()` 方法来运行，输出文件将自动被解析到一个结果目录。

下面是一个 `codeml` 典型用法的例子：

```
>>> from Bio.Phylo.PAML import codeml
>>> cml = codeml.Codeml()
>>> cml.alignment = "Tests/PAML/alignment.phymlp"
>>> cml.tree = "Tests/PAML/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")
```

已有的输出文件也可以通过模块的 `read()` 方法来解析：

```
>>> results = codeml.read("Tests/PAML/Results/codeml/codeml_NSsites_all.out")
>>> print results.get("lnL_max")
```

这个新模块的详细介绍目前在 Biopython 维基上可以看到：<http://biopython.org/wiki/PAML>

13.7 未来计划

Bio.Phylo 目前还在开发中，下面是我们可能会在将来的发布版本中添加的特性：

新方法 通常用来操作 Tree 和 Clade 对象的有用方法会首先出现在 Biopython 维基上，这样常规用户就能在我们添加到 Bio.Phylo 之前测试这些方法，看看它们是否有用：
http://biopython.org/wiki/Phylo_cookbook

Bio.Nexus port 这个模块的大部分是在 2009 年 NESCent 主办的谷歌编程夏令营中写的，作为实现 Python 对 phyloXML 数据格式（见 13.4.4）支持的一个项目。对 Newick 和 Nexus 格式的支持，已经通过导入 Bio.Nexus 模块的一部分被添加到 Bio.Phylo 使用的新类中。

目前，Bio.Nexus 包含一些还没有导入到 Bio.Phylo 类中的有用的特性——特别是，计算一致树 (consensus tree)。如果你发现某些功能 Bio.Phylo 中没有，试试在 Bio.Nexus 中能不能找到。

我们乐意接受任何增强该模块功能和使用性的建议；如果有，只需要通过邮件列表或我们的 bug 数据库让我们知道。

第 14 章使用 BIO.MOTIFS 进行模体序列分析

本章主要的介绍 Biopython 中的 Bio.motifs 包。这个包是为了方便那些需要进行模体序列分析的人们而特意提供的，所以我想你们在使用时肯定对模体序列分析的一些相关要点都很熟悉。假如在使用中遇到不清楚的地方，请您查阅 14.8 相关章节以获得有关的信息。

本章的大部分内容是介绍 Biopython 1.61 之前版本中新加入的 Bio.motifs 包，该包替代了 Biopython 1.50 版本中的 Bio.Motif 包，而 Bio.Motif 包是基于较早版本的 Biopython 中的两个模块 Bio.AlignAce 和 Bio.MEME。Bio.motifs 包较好地综合了上述的几个模块的功能，做为一个统一模块工具。

说到其他库，看到这里，你或许会对 TAMO 感兴趣，这是另一个分析模体序列的 Python 库。它能提供更多关于 *de-novo* 模体的查找方式，不过它并没有纳入到 Biopython 中，而且在商业用途上还有一些限制。

14.1 模体对象

由于我们感兴趣的是模体分析，所以我们需要先看看 Motif 对象。对此我们需要先导入 Bio.motifs 包：

```
>>> from Bio import motifs
```

然后我们可以开始创建我们第一个模体对象。我们可以从模体的实例列表中创建一个 Motif 对象，也可以通过读取模体数据库中或模体查找软件产生的文件来获得一个 Motif 对象。

```
>>> from Bio import motifs
```

14.1.1 从实例中创建一个模体

假设我们有一些 DNA 模体的实例：

```
>>> from Bio.Seq import Seq
>>> instances = [Seq("TACAA"),
...             Seq("TACGC"),
...             Seq("TACAC"),
...             Seq("TACCC"),
...             Seq("AACCC"),
...             Seq("AATGC"),
...             Seq("AATGC"),
...             ]
```

然后我们可以如下创建一个模体对象：

```
>>> m = motifs.create.instances)
```

这些实例被存储在一个名为 `m.instances` 的属性中，这个其实也就是一个 Python 的列表，只不过附加了一些功能，这些功能将在之后介绍。将这些模体对象打印出来后可以看出这些实例是从哪构建出来的。

```
>>> print m
TACAA
TACGC
TACAC
TACCC
AACCC
AATGC
AATGC
```

模体的长度像其他一些实例一些被定义为序列的长度：

```
>>> len(m)
5
```

模体对象有一个 `.counts` 属性，可以用来查看碱基在每个位置的数目。可以把这个统计表用易读的格式打印出来：

```
>>> 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
```

你也可以像使用字典一样获取这些数目：

```
>>> m.counts['A']
[3, 7, 0, 2, 1]
```

但是你也可以把它看成一个二维数列，核苷酸作为列，位置作为行：

```
>>> m.counts['T',0]
4
>>> m.counts['T',2]
2
>>> m.counts['T',3]
0
```

你还可以直接获得核苷酸数目矩阵中的列

```
>>> m.counts[:,3]
{'A': 2, 'C': 2, 'T': 0, 'G': 3}
```

除了使用核苷酸本身，你还可以使用模体碱基序列按字符排序后的核苷酸索引：

```
>>> m.alphabet
IUPACUnambiguousDNA()
>>> m.alphabet.letters
'GATC'
>>> sorted(m.alphabet.letters)
['A', 'C', 'G', 'T']
>>> m.counts['A',:]
(3, 7, 0, 2, 1)
>>> m.counts[0,:]
(3, 7, 0, 2, 1)
```

模体有一个相关联的一致序列，这个序列被定义为由 `.counts` 矩阵相应列中具有最大值的碱基，这些碱基是按模体序列排列的：

```
>>> m.consensus
Seq('TACGC', IUPACUnambiguousDNA())
```

反一致序列也一样，只不过是 `.counts` 矩阵中相应列的最小值来选：

```
>>> m.anticonsensus
Seq('GGGTG', IUPACUnambiguousDNA())
```

你也可以利用简并一致序列，用不确定核苷酸来表示序列某一位置的所有核苷酸：

```
>>> m.degenerate_consensus
Seq('WACVC', IUPACAmbiguousDNA())
```

此处，W 和 R 都是按照 IUPAC 不确定核苷酸表规定的：W 代表 A 或 T，V 代表 A，C 或 G [10]。这些简并一致序列是按照 Cavener 指定的规则 [11] 来建立的。

```
>>> r = m.reverse_complement()
>>> r.consensus
Seq('GCGTA', IUPACUnambiguousDNA())
>>> r.degenerate_consensus
Seq('GBGTW', IUPACAmbiguousDNA())
>>> print r
TTGTA
GCGTA
GTGTA
GGGTA
GGGTT
GCATT
GCATT
```

反向互补序列和简并一致序列都只在 DNA 模体中有。

14.1.2 读取模体

从实例手动创建一个模体确实有点无趣，所以用一些 I/O 函数来读写模体是很有用的。目前对于如何存储模体还没有一些真正的标准，不过有一些格式用得比其他更经常。这其中最重要的区别在于模体表示是基于实例还是某种 PWM 矩阵。

JASPAR

作为一个最流行的模体数据库 JASPAR 它不是以一系列的实例就是频率矩阵。比如，下面就是 JASPAR Arnt.sites 文件的开头和结尾行显示了老鼠螺旋 - 环 - 螺旋转录因子 Arnt 的结合位点：

```
>MA0004 ARNT 1
CACGTGatgtcctc
>MA0004 ARNT 2
CACGTGggaggtac
>MA0004 ARNT 3
CACGTGccgcgcgc
...
>MA0004 ARNT 18
AACGTGacagccctcc
>MA0004 ARNT 19
AACGTGcacatcgtcc
```



```
>MA0004 ARNT      20
aggaatCGCGTGc
```

那些用大写字母表示的序列的一部分就是被用来相互比对的模体实例。

我们可以从下面的实例创建一个 Motif 对象：

```
>>> from Bio import motifs
>>> arnt = motifs.read(open("Arnt.sites"), "sites")
```

从这个模体创建的实例存储在该模体的 `.instances` 属性：

```
>>> print arnt.instances[:3]
[Seq('CACGTG', IUPACUnambiguousDNA()), Seq('CACGTG', IUPACUnambiguousDNA()), Seq('CACGTG', IUPACUnambiguousDNA())]
>>> for instance in arnt.instances:
...     print instance
...
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
CACGTG
AACGTG
AACGTG
AACGTG
AACGTG
CGCGTG
```

这个模体的计数矩阵可以从这些实例中自动计算出来：

```
>>> 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
```

JASPAR 数据库也可以让模体像计数矩阵一样获得，不需要那些创建它们的实例。比如，下面这个 JASPAR 文件 `SRF.pfm` 包含了人类 SRF 转录因子的计数矩阵：

```
 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
```

我们可以如下为计数矩阵创建一个模体：

```
>>> srf = motifs.read(open("SRF.pfm"), "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

```

由于这个模体是由计数矩阵直接创建的，所以它没有相关的实例：

```
>>> print srf.instances
None
```

我们可以获得这两个模体的一致序列：

```
>>> print arnt.counts.consensus
CACGTG
>>> print srf.counts.consensus
GCCCATATATGG
```

MEME

MEME [12] 是一个用来在一堆相关 DNA 或蛋白质序列中发现模体的工具。它输入一组相关 DNA 或蛋白质序列，输出所要求的模体。因此和 JASPAR 文件相比，MEME 输出文件里面一般是含有多个模体。例子如下。

在输出文件的开头，有一些 MEME 生成的关于 MEME 和所用 MEME 版本的背景信息：

```

*****
MEME - Motif discovery tool
*****
MEME version 3.0 (Release date: 2004/08/18 09:07:01)
...

```

再往下，简要概括了输入的训练序列集：

```

*****
TRAINING SET
*****
DATAFILE= IN0_up800.s
ALPHABET= ACGT
Sequence name      Weight Length  Sequence name      Weight Length
-----
CH01               1.0000    800  CH02               1.0000    800
FAS1               1.0000    800  FAS2               1.0000    800
ACC1               1.0000    800  IN01               1.0000    800
OPI3               1.0000    800
*****

```

以及所使用到的命令：

```

*****
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 IN0_up800.s
...

```

接下来就是每个被发现模体的详细信息：

```
*****
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:
```

使用下面的方法来读取这个文件（以 `meme.dna.oops.txt` 存储）：

```
>>> handle = open("meme.dna.oops.txt")
>>> record = motifs.parse(handle, "meme")
>>> handle.close()
```

`motifs.parse` 命令直接读取整个文件，所以在使用后可以关闭这个文件。其中头文件信息被存储于属性中

```
>>> record.version
'3.0'
>>> record.datafile
'INO_up800.s'
>>> record.command
'meme -mod oops -dna -revcomp -nmotifs 2 -bfile yeast.nc.6.freq INO_up800.s'
>>> record.alphabet
IUPACUnambiguousDNA()
>>> record.sequences
['CH01', 'CH02', 'FAS1', 'FAS2', 'ACC1', 'INO1', 'OPI3']
```

这个数据记录是 `Bio.motifs.meme.Record` 类的一个对象。这个类继承于列表（list），所以你可以把这个 `record` 看成模体对象的一个列表：

```
>>> len(record)
2
>>> motif = record[0]
>>> print motif.consensus
TTCACATGCCGC
>>> print motif.degenerate_consensus
TTCACATGSCNC
```

除了一般的模体属性外，每个模体还同时保存着它们由 MEME 计算的各自特异信息。例如：

```
>>> motif.num_occurrences
7
>>> motif.length
12
>>> evalue = motif.evalue
>>> print "%3.1g" % evalue
0.2
>>> motif.name
'Motif 1'
```

除了像上面所做的用索引来获得相关记录，你也可以用它的名称来找到这个记录：

```
>>> motif = record['Motif 1']
```

每个模体都有一个 `.instances` 属性与在这个被发现模体中的序列实例，能够为每个实例提供一些信息：

```
>>> len(motif.instances)
7
>>> motif.instances[0]
Instance('TTCACATGCCGC', IUPACUnambiguousDNA())
>>> motif.instances[0].motif_name
'Motif 1'
>>> motif.instances[0].sequence_name
'IN01'
>>> motif.instances[0].start
620
>>> motif.instances[0].strand
'-'
>>> motif.instances[0].length
12
>>> pvalue = motif.instances[0].pvalue

>>> print "%5.3g" % pvalue
1.85e-08
```

MAST

TRANSFAC

TRANSFAC 是一个为转录因子手动创建的一个专业数据库，同时还包括染色体结合位点和 DNA 结合的描述 [27]。TRANSFAC 数据库中所用的文件格式至今还被其他工具所使用，我们下面将介绍 TRANSFAC 文件格式。

TRANSFAC 文件格式简单概括如下：

```
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
//
```

这个文件显示了模体 motif1 中 12 个核苷酸的频率矩阵。总的来说，一个 TRANSFAC 文件里面可以包含多个模体。以下是示例文件 transfac.dat 的内容：

```
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
//
```

可用如下方法读取 TRANSFAC 文件：

```
>>> handle = open("transfac.dat")
>>> record = motifs.parse(handle, "TRANSFAC")
>>> handle.close()
```

如果有总版本号的话，它是存储在 record.version 中：

```
>>> record.version
'EXAMPLE January 15, 2013'
```

每个在 record 中的模体都是 Bio.motifs.transfac.Motif 类的实例，这些实例同时继承 Bio.motifs.Motif 类和 Python 字典的属性。这些字典用双字母的键来存储关于这个模体的其他附加信息：

```
>>> motif = record[0]
>>> motif.degenerate_consensus # Using the Bio.motifs.Motif method
Seq('SRACAGGTGKYG', IUPACAmbiguousDNA())
>>> motif['ID'] # Using motif as a dictionary
'motif1'
```

TRANSFAC 文件一般比这些例子更详细，包含了许多关于模体的附加信息。表格 14.1.2 列出了在 TRANSFAC 文件常见的双字母含义：

Table 14.1: TRANSFAC 文件中常见的字段

AC	Accession numbers 序列号
AS	Accession numbers, secondary 第二序列号
BA	Statistical basis 统计依据
BF	Binding factors 结合因子
BS	Factor binding sites underlying the matrix 基于矩阵的转录结合位点
CC	Comments 注解
CO	Copyright notice 版权事项
DE	Short factor description 短因子说明
DR	External databases 外部数据库
DT	Date created/updated 创建或更新日期
HC	Subfamilies 亚家庭名称
HP	Superfamilies 超家庭名称
ID	Identifier 身份证
NA	Name of the binding factor 结合因子的名称
OC	Taxonomic classification 分类
OS	Species/Taxon 种类或分类
OV	Older version 旧版本
PV	Preferred version 首选版本
TY	Type 类型
XX	Empty line; these are not stored in the Record. 空白行; 没在记录中存储的数据

每个模体同时也有一个包含与这个模体相关参考资料的 `references` 属性，用下面的双字母键来获得：

Table 14.2: TRANSFAC 文件中用来存储参考资料的字段

RN	Reference number 参考数目
RA	Reference authors 参考资料作者
RL	Reference data 参考数据
RT	Reference title 参考标题
RX	PubMed ID

将 TRANSFAC 文件按原来格式打印出来：

```
>>> 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
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
//
```

通过用字符串形式来截取输出并且保存在文件中，你可以按 TRANSFAC 的格式导出这些模体：

```
>>> text = str(record)
>>> handle = open("mytransfacfile.dat", 'w')
>>> handle.write(text)
>>> handle.close()
```

14.1.3 模体写出

说到导出，我们可以先看看导出函数。以 JASPAR .pfm 格式导出模体文件，可以用：

```
>>> print m.format("pfm")
3      7      0      2      1
0      0      5      2      6
0      0      0      3      0
4      0      2      0      0
```

用类似 TRANSFAC 的格式导出一个模体：

```
>>> print m.format("transfac")
P0      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
//
```

你可以用 motifs.write 来写出多个模体。这个函数在使用的时候不必担心这些模体来自于 TRANSFAC 文件。比如：

```
>>> two_motifs = [arnt, srf]
>>> print motifs.write(two_motifs, 'transfac')
P0      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
//
P0      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      T
11      2      0      44      0      G
12      2      1      43      0      G
XX
//

```

14.1.4 绘制序列标识图

如果能够联网，我们可以创建一个 `weblogo`：

```
>>> arnt.weblogo("Arnt.png")
```

将得到的标识图存储成 PNG 格式。

14.2 位置权重矩阵

模体对象的 `.counts` 属性能够显示在序列上每个位置核苷酸出现的次数。我们可以把这矩阵除以序列中的实例数目来标准化这矩阵，得到每个核苷酸在序列位置上出现概率。我们把这个概率看作位置权重矩阵。不过，要知道在字面上，这个术语也可以用来说明位置特异性得分矩阵，这个我们将会在下面讨论。

通常来说，伪计数（pseudocounts）在归一化之前都已经加到每个位置中。这样可以避免在这序列上过拟合位置权重矩阵以至趋向于模体的实例的有限数量，还可以避免概率为 0。向每个位置的核苷酸添加一个固定的伪计数，可以为 `pseudocounts` 参数指定一个数值：

```

>>> 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

```

另外，`pseudocounts` 可以利用字典为每个核苷酸指定一个伪计数值。例如，由于在人类基因组中 GC 含量大概为 40%，因此可以选择下面这些伪计数值：

```

>>> 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

```

位置权重矩阵有它自己的方法计算一致序列、反向一致序列和简并一致序列：

```
>>> pwm.consensus
Seq('TACGC', IUPACUnambiguousDNA())
>>> pwm.anticonsensus
Seq('GGGTG', IUPACUnambiguousDNA())
>>> pwm.degenerate_consensus
Seq('WACNC', IUPACAmbiguousDNA())
```

应当注意到由于伪计数的原因，由位置权重矩阵计算得到的简并一致序列和由模体中实例计算得到的简并一致序列有一点不同：

```
>>> m.degenerate_consensus
Seq('WACVC', IUPACAmbiguousDNA())
```

位置权重矩阵的反向互补矩阵可以直接用 `pwm` 计算出来：

```
>>> 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
```

14.3 位置特异性得分矩阵

使用背景分布和加入伪计数的 PWM，很容易就能计算出 log-odds 比率，提供特定标记的 log odds 值，这值来自于在这个背景的模式。我们可以用在位置权重矩阵中 `.log-odds()` 方法：

```
>>> 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
```

这时我们可以更经常看到特定标记和背景下的正值和负值。0.0 意味着在模体和背景中观察到一个标记有相等的可能性。

上面是假设 A,C,G 和 T 在背景中出现的概率是相同的。那在 A,C,G 和 T 出现概率不同的情况下，为了计算特定背景下的位置特异性得分矩阵，可以使用 `background` 参数。例如，在 40%GC 含量的背景下，可以用：

```
>>> 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
```

从 PSSM 中得到的最大和最小值被存储在 `.max` 和 `.min` 属性中：

```
>>> print "%.2f" % pssm.max
6.59
>>> print "%.2f" % pssm.min
-10.85
```

在特定背景下计算平均值和标准方差使用 `.mean` 和 `.std` 方法。

```
>>> mean = pssm.mean(background)
>>> std = pssm.std(background)
>>> print "mean = %.2f, standard deviation = %.2f" % (mean, std)
mean = 3.21, standard deviation = 2.59
```

如果没有指定特定的背景，就会使用一个统一的背景。因为同 KL 散度或相对熵的值相同，所以平均值就显得特别重要，并且它也是同背景相比的模体信息含量的测量方法。由于在 Biopython 中用以 2 为底的对数来计算 log-odds 值，信息含量的单位是 bit。

`.reverse_complement`, `.consensus`, `.anticonsensus` 和 `.degenerate_consensus` 方法可以直接对 PSSM 使用。

14.4 搜索实例

模体最常用的功能就是在序列中的查找它的实例。在这节，我们会用如下的序列作为例子：

```
>>> test_seq=Seq("TACACTGCATTACAACCCAAGCATT",m.alphabet)
>>> len(test_seq)
26
```

14.4.1 搜索准确匹配实例

查找实例最简单的方法就是查找模体实例的准确匹配：

```
>>> for pos,seq in m.instances.search(test_seq):
...     print pos, seq
...
0 TACAC
10 TACAA
13 AACCC
```

我们可获得反向互补序列（找到互补链的实例）：

```
>>> for pos,seq in r.instances.search(test_seq):
...     print pos, seq
...
6 GCATT
20 GCATT
```

14.4.2 用 PSSM 得分搜索匹配实例

在模体中很容易找出相应的位置，引起对模体的高 log-odds 值：

```
>>> for position, score in pssm.search(test_seq, threshold=3.0):
...     print "Position %d: score = %.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
```

负值的位置是指在测试序列的反向链中找到的模体的实例，而且得力于 Python 的索引。在 pos 的模体实例可以用 `test_seq[pos:pos+len(m)]` 来定位，不管 pos 值是正还是负。

你可能注意到阈值参数，在这里随意地设为 3.0。这里是 \log_2 ，所以我们现在开始寻找那些在模体中出现概率为背景中出现概率 8 倍序列。默认的阈值是 0.0，在此阈值下，会把所有比背景中出现概率大的模体实例都找出来。

```
>>> 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)
```

通常来说，上述是计算 PSSM 得分的最快方法。这些得分只能由前导链用 `pssm.calculate` 计算得到。为了得到互补链的 PSSM 值，你可以利用 PSSM 的互补矩阵：

```
>>> 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)
```

14.4.3 选择得分阈值

如果不想刚才那么随意设定一个阈值，你可以探究一下 PSSM 得分的分布。由于得分的空间分布随着模体长度而成倍增长，我们用一个近似于给定精度值来计算，如此可使计算成本更容易控制：

```
>>> distribution = pssm.distribution(background=background, precision=10**4)
```

`distribution` 对象可以用来决定许多不同的阈值。我们可以指定一个需要的的假阳性率（找到一个由序列在此背景下产生的模体实例的概率）：

```
>>> threshold = distribution.threshold_fpr(0.01)
>>> print "%5.3f" % threshold
4.009
```

或者假阴性率（找不到模体产生的实例概率）：

```
>>> threshold = distribution.threshold_fnr(0.1)
>>> print "%5.3f" % threshold
-0.510
```

或者一个阈值（近似），满足假阳性率和假阴性率之间的关系 ($\text{fnr}/\text{fpr} \approx t$)：

```
>>> threshold = distribution.threshold_balanced(1000)
>>> print "%5.3f" % threshold
6.241
```

或者一个阈值能够大体满足假阳性率和信息含量的 \log 值之间的相等关系（与 Hertz 和 Stormo 的 Patser 软件所用的一样）：

```
>>> threshold = distribution.threshold_patser()
>>> print "%5.3f" % threshold
0.346
```

比如在我们这个模体中，当以 1000 比率的平衡阈值查找实例，你可以得到一个让你获得相同结果的阈值（对这个序列来说）。

```
>>> 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
```

14.5 模体对象自身相关的位置特异性得分矩阵

为了更好的利用 PSSMs 来查找潜在的 TFBSs，每个模体都同位置权重矩阵和位置特异性得分矩阵相关联。用 Arnt 模体来举个例子：

```
>>> from Bio import motifs
>>> handle = open("Arnt.sites")
>>> 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

>>> 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

>>> 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
```

在这出现的负无穷大是由于在频率矩阵中相关项的值为 0，并且我们默认使用 0 作为伪计数：

```
>>> for letter in "ACGT":
...     print "%s: %4.2f" % (letter, motif.pseudocounts[letter])
...
A: 0.00
C: 0.00
```

```
G: 0.00
T: 0.00
```

如果你更改了 `.pseudocounts` 属性，那么位置频率矩阵和位置特异性得分矩阵就都会自动重新计算：

```
>>> motif.pseudocounts = 3.0
>>> for letter in "ACGT":
...     print "%s: %.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

>>> 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
```

如果你想对 4 个核苷酸使用不同的伪计数，可以使用字典来设定 4 个核苷酸的 `pseudocounts`。把 `motif.pseudocounts` 设为 `None` 会让伪计数重置为 0 的默认值。

位置特异性得分矩阵依赖于一个默认均一的背景分布：

```
>>> for letter in "ACGT":
...     print "%s: %.2f" % (letter, motif.background[letter])
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

同样，如果你更改了背景分布，位置特异性得分矩阵也会重新计算：

```
>>> 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
```

把 `motif.background` 设为 `None` 后会将其重置为均一的分布。

```
>>> motif.background = None
>>> for letter in "ACGT":
...     print "%s: %.2f" % (letter, motif.background[letter])
...
A: 0.25
C: 0.25
G: 0.25
T: 0.25
```

如果你把 `motif.background` 设为一个单一值，这个值将会被看成是 GC 含量：

```
>>> motif.background = 0.8
>>> for letter in "ACGT":
...     print "%s: %.2f" % (letter, motif.background[letter])
...
A: 0.10
C: 0.40
G: 0.40
T: 0.10
```

应当注意到你能够在当前计算背景下计算 PSSM 的平均值：

```
>>> print "%f" % motif.pssm.mean(motif.background)
4.703928
```

它的标准方差也是一样：

```
>>> print "%f" % motif.pssm.std(motif.background)
3.290900
```

和它的分布：

```
>>> distribution = motif.pssm.distribution(background=motif.background)
>>> threshold = distribution.threshold_fpr(0.01)
>>> print "%f" % threshold
3.854375
```

请注意，每当你调用 `motif.pwm` 或 `motif.pssm`，位置权重矩阵和位置特异性得分矩阵都会重新计算。如果看重速度并且需要重复用到 PWM 或 PSSM 时，你可以把他们保存成变量，如下所示：

```
>>> pssm = motif.pssm
```

14.6 模体比较

当有多个模体时，我们就会想去比较它们。

在我们开始比较之前，应当要指出模体的边界通常比较模糊。这也就是说我们需要比较不同长度的模体，因此这些比较也要涉及到相关的比对。所以我们需要考虑两个东西：

- 模体比对
- 比较比对后模体的相关函数

为了比对模体，我们使用 PSSMs 的不含间隔的比对，并且用 0 来代替矩阵开始和结束位置缺失的列。这说明我们能够有效地利用背景分布来代替 PSSM 中缺失的列。距离函数然后可以返回模体间最小的距离，以及比对中相应的偏移量。

举个例子，先导入和测试模体 `m` 相似的模体：

```
>>> m_reb1 = motifs.read(open("REB1.pfm"), "pfm")
>>> m_reb1.consensus
Seq('GTTACCCGG', IUPACUnambiguousDNA())
>>> 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

为了让模体能够进行相互比较，选择和模体 `m` 相同伪计数和背景值：

```
>>> 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
```

我们将用皮尔逊相关 (Pearson correlation) 来比较这些模体。由于我们想要让它偏向于一个距离长度，我们实际上取 $1 - r$ ，其中 r 是皮尔逊相关系数 (Pearson correlation coefficient, PCC)：

```
>>> distance, offset = pssm.dist_pearson(pssm_reb1)
>>> print "distance = %5.3g" % distance
distance = 0.239
>>> print offset
-2
```

这意味着模体 `m` 和 `m_reb1` 间最佳 PCC 可以从下面的比对中获得：

```
m:      bbTACGCbb
m_reb1: GTTACCCGG
```

其中 `b` 代表背景分布。PCC 值大概为 $1 - 0.239 = 0.761$ 。

14.7 查找 *De novo* 模体

如今，Biopython 对 *De novo* 模体查找的支持是有限的。也就是说，我们支持 AlignAce 和 MEME 的运行和读取。由于模体查找工具如雨后春笋般出现，所以很欢迎新的分析程序加入进来。

14.7.1 MEME

假设用 MEME 以你喜欢的参数设置来跑序列，并把结果保存在文件 `meme.out` 中。你可以用以下的命令来得到 MEME 输出的模体：

```
>>> from Bio import motifs
>>> motifsM = motifs.parse(open("meme.out"), "meme")

>>> motifsM
[<Bio.motifs.meme.Motif object at 0xc356b0>]
```

除了最想要的一系列模体外，结果中还包含了很多有用的信息，可以通过那些一目了然的属性名获得：

- `.alphabet`
- `.datafile`
- `.sequence_names`
- `.version`
- `.command`

由 MEME 解析得到的模体可以像平常的模体对象（有实例）一样处理，它们也提供了一些额外的功能，可以为实例增加额外的信息。

```
>>> motifsM[0].consensus
Seq('CTCAATCGTA', IUPACUnambiguousDNA())
>>> motifsM[0].instances[0].sequence_name
'SEQ10;'
>>> motifsM[0].instances[0].start
3
>>> motifsM[0].instances[0].strand
'+'

>>> motifsM[0].instances[0].pvalue
8.71e-07
```

14.7.2 AlignAce

我们可以用 AlignACE 程序实现类似的效果。假如，你把结果保存在 alignace.out 文件中。你可以用下面的代码读取结果：

```
>>> from Bio import motifs
>>> motifsA = motifs.parse(open("alignace.out"), "alignace")
```

同样，你的模体也和正常的模体对象有相同的属性：

```
>>> motifsA[0].consensus
Seq('TCTACGATTGAG', IUPACUnambiguousDNA())
```

事实上，你甚至可以观察到，AlignAce 找到了一个和 MEME 非常相似的模体。下面只是 MEME 模体互补链的一个较长版本：

```
>>> motifsM[0].reverse_complement().consensus
Seq('TACGATTGAG', IUPACUnambiguousDNA())
```

如果你的机器上安装了 AlignAce，你可以直接从 Biopython 中运行 AlignAce。下面就是一个如何运行 AlignAce 的简单例子（其他参数可以用关键字参数来调用）：

```
>>> command="/opt/bin/AlignACE"
>>> input_file="test.fa"
>>> from Bio.motifs.applications import AlignAceCommandline
>>> cmd = AlignAceCommandline(cmd=command, input=input_file, gcback=0.6, numcols=10)
>>> stdout, stderr= cmd()
```

由于 AlignAce 把所有的结果输出到标准输出，所以你可以通过读取结果的第一部分来获得模体：

```
>>> motifs = motifs.parse(stdout, "alignace")
```

14.8 相关链接

- [Sequence motif](#) in wikipedia
- [PWM](#) in wikipedia
- [Consensus sequence](#) in wikipedia
- [Comparison of different motif finding programs](#)

14.9 旧版 Bio.Motif 模块

本章剩下部分将介绍 Biopython 1.61 版本前的 Bio.Motifs 模块，该模块取代了 Biopython 1.50 版本中基于两个早期 Biopython 模块—— Bio.AlignAce 和 Bio.MEME 的 Bio.Motif 模块。

为了平滑的过渡，早期的 Bio.Motif 模块将会和它的取代者 Bio.Motifs 一同维护到至少发行两个版本，并且持续至少一年。

14.9.1 模体对象

由于我们对模体分析感兴趣，不过让我们首先看看 Motif 对象。第一步要先导入模体库：

```
>>> from Bio import Motif
```

然后可以开始创建第一个模体对象。创建一个 DNA 模体：

```
>>> from Bio.Alphabet import IUPAC
>>> m = Motif.Motif(alphabet=IUPAC.unambiguous_dna)
```

现在这里面什么也没有，往新建的模体加入一些序列：

```
>>> from Bio.Seq import Seq
>>> m.add_instance(Seq("TATAA",m.alphabet))
>>> m.add_instance(Seq("TATTA",m.alphabet))
>>> m.add_instance(Seq("TATAA",m.alphabet))
>>> m.add_instance(Seq("TATAA",m.alphabet))
```

现在我们有了一个完整的 Motif 实例，我们可以试着从中获取一些基本信息。先看看长度和一致序列：

```
>>> len(m)
5
>>> m.consensus()
Seq('TATAA', IUPACUnambiguousDNA())
```

对于 DNA 模体，我们还可以获得一个模体的反向互补序列：

```
>>> m.reverse_complement().consensus()
Seq('TTATA', IUPACUnambiguousDNA())
>>> for i in m.reverse_complement().instances:
...     print i
TTATA
TAATA
TTATA
TTATA
```

我们也可以简单的调取模体的信息容量：

```
>>> print "%.2f" % m.ic()
5.27
```

这给我们提供了模体中信息容量的比特数，这指出和背景有多少不同。

展示模体最常用的就是 PWM（位置权重矩阵）。它概括了在模体上任意位置出现一个符号（这里指核苷酸）的概率。这个可以用 .pwm() 方法来计算：

```
>>> m.pwm()
[{'A': 0.05, 'C': 0.05, 'T': 0.85, 'G': 0.05},
```

```
{'A': 0.85, 'C': 0.05, 'T': 0.05, 'G': 0.05},
{'A': 0.05, 'C': 0.05, 'T': 0.85, 'G': 0.05},
{'A': 0.65, 'C': 0.05, 'T': 0.25, 'G': 0.05},
{'A': 0.85, 'C': 0.05, 'T': 0.05, 'G': 0.05}]
```

模体的 PWM 中的概率是基于实例中的计数，但我们发现，虽然模体中没有出现 G 和 C，可是它们的概率仍然是非 0 的。这主要是因为有伪计数的存在，简单地说，就是一种常用的方式来承认我们认知的不完备以及为了避免使用 0 进行对数运算而出现的技术问题。

我可以调整伪计数添加到模体对象两个属性的方式。`.background` 是我们假设代表背景分布的所有字符的概率分布，是非模体序列（通常基于各自基因组的 GC 含量）。在模体创建的时候，就默认的设置为一个统一分布：

```
>>> m.background
{'A': 0.25, 'C': 0.25, 'T': 0.25, 'G': 0.25}
```

另一个就是 `.beta`，这个参数可以说明我们应该给伪计数设定为何值。默认设定为 1.0。

```
>>> m.beta
1.0
```

所以输入伪计数的总量等于一个实例的输入总量。

使用背景分布和附加了伪计数的 `pwm`，可以很容易的计算 `log-odd` 比率，这告诉我们在背景下，一个来自模体特定碱基的 `log-odd` 值。我们可以使用 `.log_odds()` 方法：

```
>>> m.log_odds()
[{'A': -2.3219280948873622,
 'C': -2.3219280948873622,
 'T': 1.7655347463629771,
 'G': -2.3219280948873622},
 {'A': 1.7655347463629771,
 'C': -2.3219280948873622,
 'T': -2.3219280948873622,
 'G': -2.3219280948873622},
 {'A': -2.3219280948873622,
 'C': -2.3219280948873622,
 'T': 1.7655347463629771,
 'G': -2.3219280948873622},
 {'A': 1.3785116232537298,
 'C': -2.3219280948873622,
 'T': 0.0,
 'G': -2.3219280948873622},
 {'A': 1.7655347463629771,
 'C': -2.3219280948873622,
 'T': -2.3219280948873622,
 'G': -2.3219280948873622}]
```

此处，我们可以看出如果模体中的碱基比背景中出现频率更高，其值为正值，反之则为负值。0.0 说明在背景和模体中出现的概率是相同的（如第二个位置的“T”）。

14.9.1.1 模体读写

手动从实例创建一个模体确实没什么技术含量，所以很有必要有一些读写功能来读取和写出模体。对于如何存储模体还没有一个固定的标准，但是有一些格式比其他格式更流行。这些格式的主要区别在于模体的创建是基于实例还是是一些 PWM 矩阵。其中一个最流行的模体数据库就是 [JASPAR](#)，该数据库保存了上述两种类型的格式，所以让我们看看是如何从实例中导入 JASPAR 模体：

```
>>> from Bio import Motif
>>> arnt = Motif.read(open("Arnt.sites"), "jaspar-sites")
```

从一个计数矩阵中导入：

```
>>> srf = Motif.read(open("SRF.pfm"), "jaspar-pfm")
```

arnt 和 srf 模体可以为我们做相同的事情，但是它们使用不同的内部表现形式来展现模体。我们可以用 has_counts 和 has_instances 属性来区分它们：

```
>>> arnt.has_instances
True
>>> srf.has_instances
False
>>> srf.has_counts
True

>>> srf.counts
{'A': [2, 9, 0, 1, 32, 3, 46, 1, 43, 15, 2, 2],
 'C': [1, 33, 45, 45, 1, 1, 0, 0, 0, 1, 0, 1],
 'G': [39, 2, 1, 0, 0, 0, 0, 0, 0, 0, 44, 43],
 'T': [4, 2, 0, 0, 13, 42, 0, 45, 3, 30, 0, 0]}
```

对于模体的不同表现形式，可以用转换功能来实现相互转换：

```
>>> arnt.make_counts_from_instances()
{'A': [8, 38, 0, 0, 0, 0],
 'C': [32, 0, 40, 0, 0, 0],
 'G': [0, 2, 0, 40, 0, 40],
 'T': [0, 0, 0, 0, 40, 0]}

>>> srf.make_instances_from_counts()
[Seq('GGGAAAAAAGG', IUPACUnambiguousDNA()),
 Seq('GGCCAAATAAGG', IUPACUnambiguousDNA()),
 Seq('GACCAAATAAGG', IUPACUnambiguousDNA()),
 ...]
```

在这里需要注意的是 make_instances_from_counts() 方法创建的是假实例，因为按照相同的 pwm 能够得到许多不同的实例，所以不能反过来重建原来的矩阵。不过这对我们利用 PWM 来展现模体没有什么影响，但从基于计数的模体中导出实例时要小心。

说到导出，让我们看看导出函数。我们可以按 fasta 的格式导出：

```
>>> print m.format("fasta")
>instance0
TATAA
>instance1
TATTA
>instance2
TATAA
>instance3
TATAA
```

或者是按 TRANSFAC 样的矩阵格式导出（能被一些处理软件识别）

```
>>> print m.format("transfac")
XX
TY Motif
ID
BF undef
```

```

P O G A T C
01 0 0 4 0
02 0 4 0 0
03 0 0 4 0
04 0 3 1 0
05 0 4 0 0
XX

```

最后，如果能够联网，我们可以创建一个 `weblogo`：

```
>>> arnt.weblogo("Arnt.png")
```

我们可以把得到的标识图以 `png` 的格式保存到特定的文件中。

14.9.2 查找实例

模体中最常用的就是在一些序列中查找实例。为解释这部分，我们将手动创建一个如下的序列：

```
test_seq=Seq("TATGATGTAGTATAATATAATTATAA",m.alphabet)
```

查找实例最简单的方法就是在模体中查找具体匹配的实例：

```
>>> for pos,seq in m.search_instances(test_seq):
...     print pos,seq.tostring()
...
10 TATAA
15 TATAA
21 TATAA

```

对于互补序列，也可以用相同的方法（为了找到互补链上的实例）：

```
>>> for pos,seq in m.reverse_complement().search_instances(test_seq):
...     print pos,seq.tostring()
...
12 TAATA
20 TTATA

```

提高模体的 `log-odds` 值能让查为位置更加简单：

```
>>> for pos,score in m.search_pwm(test_seq,threshold=5.0):
...     print pos,score
...
10 8.44065060871
-12 7.06213898545
15 8.44065060871
-20 8.44065060871
21 8.44065060871

```

你可能注意到阈值参数，在这里随意地设为 5.0。按 \log_2 来算，我们应当查找那些在模体中出现概率为背景中出现概率 32 倍的序列。默认的阈值是 0.0，在些阈值下，会把所有比背景中出现概率大的模体实例都找出来。

如果不想那么随意的选择一个阈值，你可以研究一下 `Motif.score_distribution` 类，它为模体提供一个相应的得分分布。由于得分的空间分布随着模体长度而成倍增长，我们正用一个近似于给定精度值计算，从而使计算成本易于控制：

```
>>> sd = Motif.score_distribution(m,precision=10**4)
```

上面那个 `sd` 对象可以用来决定许多不同的阈值。

我们可以设定一个需要的假阳性率（找到一个由此序列在这个背景下产生的模体实例的概率）：

```
>>> sd.threshold_fpr(0.01)
4.3535838726139886
```

或者假阴性率（找不到模体产生的实例的概率）：

```
>>> sd.threshold_fnr(0.1)
0.26651713652234044
```

或者一个阈值（近似），满足假阳性率和假阴性率之间的关系（ $\text{fnr}/\text{fpr} \approx t$ ）：

```
>>> sd.threshold_balanced(1000)
8.4406506087056368
```

或者一个阈值能够大体满足假阳性率和信息含量的 \log 值之间的相等关系（像 Hertz 和 Stormo 的 Patser 软件所用的一样）：

在我们这个例子中，当以 1000 比率的平衡阈值查找实例时，你可以得到一个让你获得相同结果（对于这个序列来说）的阈值：

```
>>> for pos,score in m.search_pwm(test_seq,threshold=sd.threshold_balanced(1000)):
...     print pos,score
...
10 8.44065060871
15 8.44065060871
-20 8.44065060871
21 8.44065060871
```

14.9.3 模体比较

当有多个模体时，我们就会想去比较他们。对此，`Bio.Motif` 有三种不同的方法来进行模体比较。

在我们开始比较之前，应当指出模体的边界通常是相当模糊的。也就是说我们经常需要比较不同长度的模体，因此这些比较涉及到相关的比对。所以我们需要考虑两个要点：

- 模体比对
- 比较比对后模体的相关函数

在 `Bio.Motif` 中有三种比较方法，这些方法都是基于来源于模体比对的想法，而采用不同方式。简单来说，我们使用不含间隔的 PSSMs 比对，并且用 0 来代替矩阵同背景相比，在开始和结束位置出现缺失的列。这三种比较方法都可以解释成距离估量，但是只有一个（`dist-dpq`）满足三角不等式。这些方法都返回距离的最小值和模体相应的偏移量。

为了展示这些比较功能是如何实现的，导入和测试模体 `m` 相似的其他模体：

```
>>> ubx=Motif.read(open("Ubx.pfm"),"jaspar-pfm")
<Bio.Motif.Motif.Motif object at 0xc29b90>
>>> ubx.consensus()
Seq('TAAT', IUPACUnambiguousDNA())
```

第一个展示的功能是基于皮尔逊相关（Pearson correlation）的。因为我们想让它类似于一个距离估量，所以我们实际上取 $1 - r$ ，其中的 r 是皮尔逊相关系数（Pearson correlation coefficient，PCC）：

```
>>> m.dist_pearson(ubx)
(0.41740393308237722, 2)
```

这意味着模体 `m` 和 `Ubx` 间最佳的 PCC 可以从下面的比对中获得：


```
bbTAAT
TATAAb
```

其中 `b` 代表背景分布。PCC 值大概为 $1-0.42=0.58$ 。如果我们尝试计算 `Ubx` 模体的互补序列：

```
>>> m.dist_pearson(ubx.reverse_complement())
(0.25784180151584823, 1)
```

我们可以发现更好的 PCC 值（大概为 0.75），并且比对也是不同的：

```
bATTA
TATAA
```

还有两个其他的功能函数：`dist_dpq`，这是基于 Kullback-Leibler 散度的真正度量（满足三角不等式）。

```
>>> m.dist_dpq(ubx.reverse_complement())
(0.49292358382899853, 1)
```

还有 `dist_product` 方法，它是基于概率的方法，这概率可以看成是两个模体独立产生两个相同实例的概率。

```
>>> m.dist_product(ubx.reverse_complement())
(0.16224587301064275, 1)
```

14.9.4 *De novo* 模体查找

目前，Biopython 对 *de novo* 模体查找只有一些有限的支持。也就是说，我们只支持 AlignAce 和 MEME 的运行和读取。由于现模体查找工具发展如雨后春笋般，我们很欢迎有新的贡献者加入。

14.9.4.1 MEME

假如你以中意的参数用 MEME 来跑你自己的序列，并把得到的结果保存在 `meme.out` 文件中。你可以用以下代码读取 MEME 产生的文件获得那些模体：

```
>>> motifsM = list(Motif.parse(open("meme.out"), "MEME"))
>>> motifsM
[<Bio.Motif.MEMEMotif.MEMEMotif object at 0xc356b0>]
```

除了那一系列想要的模体外，结果对象中还有很多有用的信息，可以用那些一目了然的属性名来获取：

- `.alphabet`
- `.datafile`
- `.sequence_names`
- `.version`
- `.command`

MEME 解析器得到的模体可以像通常模体（含有实例）一样进行处理，它们也可以通过对实例添加附加信息而提供一些额外的功能。

```
>>> motifsM[0].consensus()
Seq('CTCAATCGTA', IUPACUnambiguousDNA())

>>> motifsM[0].instances[0].pvalue
8.71e-07
```

```
>>> motifsM[0].instances[0].sequence_name
'SEQ10;'
>>> motifsM[0].instances[0].start
3
>>> motifsM[0].instances[0].strand
'+'
```

14.9.4.2 AlignAce

对于 AlignACE 程序也可以做相同的事情。假如你把结果存储于文件 alignace.out 文件中。你可以用以下代码读取结果：

```
>>> motifsA=list(Motif.parse(open("alignace.out"),"AlignAce"))
```

同样，得到的模体也和平常的模体一样：

```
>>> motifsA[0].consensus()
Seq('TCTACGATTGAG', IUPACUnambiguousDNA())
```

事实上，你甚至可以发现 AlignAce 和 MEME 得到的模体十分相似，只不过 AlignAce 模体是 MEME 模体反向互补序列的加长版本而已：

```
>>> motifsM[0].reverse_complement().consensus()
Seq('TACGATTGAG', IUPACUnambiguousDNA())
```

如果你的机器上安装了 AlignAce，你也可以直接从 Biopython 中启动。下面就是一个如何启动的小例子（其他参数可以用关键字参数指定）：

```
>>> command="/opt/bin/AlignACE"
>>> input_file="test.fa"
>>> from Bio.Motif.Applications import AlignAceCommandline
>>> cmd = AlignAceCommandline(cmd=command,input=input_file,gcbac=0.6,numcols=10)
>>> stdout,stderr= cmd()
```

由于 AlignAce 把结果打印到标准输出，因此你可以通过读取结果的第一部分来获得你想要的模体：

```
motifs=list(Motif.parse(stdout,"AlignAce"))
```

第 15 章聚类分析

聚类分析是根据元素相似度，进行分组的过程。在生物信息学中，聚类分析广泛用于基因表达数据分析，用来对具有相似表达谱的基因归类；从而鉴定功能相关的基因，或预测未知基因的功能。

Biopython 中的 `Bio.Cluster` 模块提供了常用的聚类算法。虽然 `Bio.Cluster` 被设计用于基因表达数据，它也可用于其他类型数据的聚类。`Bio.Cluster` 和其使用的 C 聚类库的说明见 De Hoon *et al.* [14]。

`Bio.Cluster` 包含了以下四种聚类算法：

- 系统聚类（成对重心法，最短距离，最大距离和平均连锁法）；
- k -means, k -medians, 和 k -medoids 聚类；
- 自组织映射（Self-Organizing Maps）；
- 主成分分析

数据表示法

用于聚类的输入为一个 $n \times m$ 的 Python 数值矩阵 `data`。在基因表达数据聚类中，每一行表示不同的基因，每一列表示不同的实验条件。`Bio.Cluster` 既可以针对每行（基因），也可以针对每列（实验条件）进行聚类。

缺失值

在芯片实验中，经常会有些缺失值，可以用一个额外的 $n \times m$ Numerical Python 整型矩阵 `mask` 表示。例如 `mask[i,j]==0`，表示 `data[i,j]` 是个缺失值，并且在分析中被忽略。

随机数生成器

k -means/medians/medoids 聚类和 Self-Organizing Maps (SOMs) 需要调用随机数生成器。在 `Bio.Cluster` 中，正态分布随机数生成器的算法是基于 L'Ecuyer [25]，二项分布的随机数生成器算法是基于 Kachitvichyanukul and Schmeiser [23] 开发的 BTPE 算法。随机数生成器在调用时会首先进行初始化。由于随机数生成器使用了两个乘同余发生器（multiplicative linear congruential generators），所以初始化时需要两个整型的种子。这两个种子可以调用系统提供的 `rand`（C 标准库）函数生成。在 `Bio.Cluster` 中，我们首先调用 `srand` 使用以秒为单位的时间戳的值初始值，再用 `rand` 随机产生两个随机数作为种子来产生正态分布的随机数。

15.1 距离函数

为了对元素根据相似度进行聚类，第一步需要定义相似度。Bio.Cluster 提供了八种不同的距离函数来衡量相似度或者距离，分别用不同的字母代表：

- 'e': Euclidean 距离;
- 'b': City-block 距离.
- 'c': Pearson 相关系数;
- 'a': Pearson 相关系数的绝对值;
- 'u': Uncentered Pearson correlation (相当于两个数据向量的夹角余弦值)
- 'x': uncentered Pearson correlation 的绝对值;
- 's': Spearman's 秩相关系数;
- 'k': Kendall's τ .

前两个距离函数满足三角形的两边和大于第三边的特点：

$$d(\underline{u}, \underline{v}) \leq d(\underline{u}, \underline{w}) + d(\underline{w}, \underline{v}) \text{ for all } \underline{u}, \underline{v}, \underline{w},$$

所以称之为 *metrics*。在任何语言中，这个意味着两点之间直线最短。

剩余的六种距离函数跟相关系数有关，距离 d 是由相关性 r 确定： $d=1-r$ 。请注意这类距离函数是 *semi-metrics*，因此不满足三角形的两边之和大于第三边的性质。例如

$$\begin{aligned}\underline{u} &= (1, 0, -1); \\ \underline{v} &= (1, 1, 0); \\ \underline{w} &= (0, 1, 1);\end{aligned}$$

通过计算 Pearson 距离，可以得到 $d(u, w) = 1.8660$ ，而 $d(u, v) + d(v, w) = 1.6340$ 。

Euclidean 距离

在 Bio.Cluster 中，Euclidean 距离被定义为

$$d = \frac{1}{n} \sum_{i=1}^n (x_i - y_i)^2.$$

求和时只考虑 x_{i*} 和 y_{i*} 都存在的值，分母 n 也相应的做出调整。当分析表达谱数据时，由于 x_{i*} 和 y_{i*} 会直接相减，因此在使用 Euclidean 距离前，请对表达谱数据标准化处理。

City-block distance

city-block distance 也称之为 Manhattan 距离，跟 Euclidean 距离有一定的相似性。Euclidean 距离表示的是两点间最短的距离，而 city-block 距离则是两点在所有维度中距离的和。由于基因表达的数据经常会有缺失数据，在 Bio.Cluster 中，city-block 距离定义为总距离除以总维度：

$$d = \frac{1}{n} \sum_{i=1}^n |x_i - y_i|.$$

City-block distance 类似于当你在从城市里一个位置到另一个位置时，所经过街道的距离。而在 Euclidean 距离中，表达谱的数据会直接相减，因此必须先对数据进行标准化。

Pearson 相关系数

Pearson 相关系数定义为：

$$r = \frac{1}{n} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{\sigma_x} \right) \left(\frac{y_i - \bar{y}}{\sigma_y} \right),$$

其中 \bar{x} , \bar{y} 分别是 x 和 y 的样品均值, σ_x^* , σ_y^* 是 x 和 y 的样品标准差. Pearson 相关系数是用于测量 x 和 y 散点图对直线的拟合程度。如果所有的点都在直线上，那么 Pearson 相关系数为 +1 or -1, 取决于直线的斜率是正还是负。如果 Pearson 相关系数等于 0，表明 x 和 y 之间没有相关性。

Pearson distance 定义为

$$d_P \equiv 1 - r.$$

由于 Pearson 相关性的值介于 -1 和 1 之间, Pearson 距离的范围为 0 和 2 之间.

Absolute Pearson correlation

通过对 Pearson 相关系数取绝对值，可以得到一个 0 和 1 之间的数。如果绝对值是 1，所有的点都位于一条斜率为正或负直线上。当绝对值为 0 时，表明 x 和 y 没有相关性。

对应的距离定义为：

$$d_A \equiv 1 - |r|,$$

其中 r 是 Pearson 相关系数. 由于 Pearson 的相关系数的绝对值介于 0 和 1 之间, 对应的距离也位于 0 和 1 之间。

在基因表达数据分析中，应当注意，当相关性的绝对值等于 1 时，表明两组基因的表达情况完全一样或者完全相反。

Uncentered correlation (夹角余弦)

在某些情况下，使用 *uncentered correlation* 比常规的 Pearson 相关系数更合适。uncentered correlation 定义为：

$$r_U = \frac{1}{n} \sum_{i=1}^n \left(\frac{x_i}{\sigma_x^{(0)}} \right) \left(\frac{y_i}{\sigma_y^{(0)}} \right),$$

其中

$$\begin{aligned} \sigma_x^{(0)} &= \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2}, \\ \sigma_y^{(0)} &= \sqrt{\frac{1}{n} \sum_{i=1}^n y_i^2}. \end{aligned}$$

$$= \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2; \sigma_y^{(0)}} \\ = \sqrt{\frac{1}{n} \sum_{i=1}^n y_i^2}.$$

这个公式同 Pearson 相关系数的公式形式一样，只是把样本均值 \bar{x} 设为 0。uncentered correlation 适用于表达量基准为 0 的情况。例如，在对基因表达分析中，使用比值对数时，当 log-ratio 等于 0 表明红绿信号强度相等，也意味着实验处理不影响基因的表达量。

uncentered correlation 系数对应的距离计算方法为：

$$d_U \equiv 1 - r_U,$$

其中 r_U 是 uncentered 相关性系数。由于 uncentered 系数位于 -1 和 1 之间，对应的距离范围为 0 与 2 之间。

由于 uncentered 相关系数值等同于两个数据向量在 n 维空间里的夹角余弦，因此也常称为夹角余弦。

Absolute uncentered correlation

与 Pearson 相关性类似，也可以用 uncentered correlation 的绝对值来定义距离：

$$d_{AU} \equiv 1 - |r_U|,$$

其中 r_U 是 uncentered 相关系数。由于 uncentered 相关系数的绝对值位于 0 和 1 之间，对应的距离也为位于 0 和 1 之间。

从几何学上来讲，uncentered 相关系数的绝对值等于两个数据所在向量的支持线（supporting lines）的角度余弦值（即不考虑向量的方向性）。

Spearman rank correlation

Spearman 秩相关系数是一种非参的相关性测量方法，对于数据中的离群点，比 Pearson 相关系数有更好的稳健性。

为了计算 Spearman 秩相关系数，首先对每个数据集里的数据按值排序，得到每个数据的对应的秩。然后，计算对两个数据的秩集合计算 Pearson 相关系数，得到 Spearman 的相关系数。

同 Pearson 相关性类似，Spearman 秩相关系数对应的距离定义为：

$$d_S \equiv 1 - r_S,$$

其中 r_S 是 Spearman 秩相关系数。

Kendall's τ

Kendall's τ 是另一个非参的计算相关性的方法。它同 Spearman 秩相关系数类似，但它不对数据进行排序，而是使用相对秩来计算 τ (see Snedecor & Cochran [29])。

Kendall's τ 对应的距离计算为：

$$d_K \equiv 1 - \tau.$$

因为 Kendall's τ 位于 -1 和 1 之间, 对应的距离位于 0 和 2 之间。

Weighting

对于 Bio.Cluster 中大部分距离函数，都可以使用加权向量。加权向量包含着数据集中每个元素的权重。如果元素 i 的权重为 w_{*i} ，那么将会认为该元素出现了 w_{*i} 次。权重值可以不为整数。对于 Spearman 秩相关系数和 Kendall's τ ，权重没有太大的意义，因此不适用于这两个函数。

计算距离矩阵

距离矩阵是 data 中，所有元素的两两间的距离的平方矩阵，可以用 Bio.Cluster 模块中 distancematrix 函数计算：

```
>>> from Bio.Cluster import distancematrix
>>> matrix = distancematrix(data)
```

其中，包含以下参数：

- data (必选) 包含所有元素的矩阵
- mask (默认: None) 缺失数据矩阵。若 mask[i,j]==0, 则 data[i,j] 缺失。若 mask==None, 表明没有缺失数据。
- weight (默认: None) 权重矩阵。若 weight==None, 则假设所有的数据使用相同的权重。
- transpose (默认: 0) 选择使用 data 中的行 (transpose==0), 或者列 (transpose==1) 来计算距离。
- dist (默认: 'e', Euclidean distance) 选择距离函数 (具体见 15.1)。

为了节省内存，函数返回的距离矩阵是一个一维数组的列表。每行的列数等于行号。因此，第一行有 0 个元素。例如：

```
[array([]),
 array([1.]),
 array([7., 3.]),
 array([4., 2., 6.])]
```

对应的距离矩阵为：

$$\begin{pmatrix} 0 & 1 & 7 & 4 \\ 1 & 0 & 3 & 2 \\ 7 & 3 & 0 & 6 \\ 4 & 2 & 6 & 0 \end{pmatrix}.$$

15.2 计算类的相关性质

计算类中心

类中心可以定义为该类中在每个维度上所有元素的平均值或者中值，可以用 `Bio.Cluster` 中的 `clustercentroids` 函数计算：

```
>>> from Bio.Cluster import clustercentroids
>>> cdata, cmask = clustercentroids(data)
```

包含了以下参数：

- `data` (必选) 包含所有元素的矩阵。
- `mask` (默认: `None`) 缺失数据矩阵。若 `mask[i,j]==0`, 则 `data[i,j]` 缺失。若 `mask==None`, 则明没有缺失数据。
- `clusterid` (默认: `None`) 一个表示每个元素的所属类的整型向量。如果 `clusterid` 是 `None`, 表明所有的元素属于相同的类。
- `method` (默认: `'a'`) 指定使用算术平方根 (`method=='a'`) 或者中值 (`method=='m'`) 来计算类中心。
- `transpose` (默认: `0`) 选择使用 `data` 中的行 (`transpose==0`), 或者列 (`transpose==1`) 来计算类中心。

这个函数返回值为元组 (`cdata`, `cmask`)。类中心的数据存储在一个二维的 Numerical Python 数组 `cdata` 中, 缺失值的结果存储在二维的 Numerical Python 整型数组 `cmask` 中。当 `transpose = 0` 时, 这两个数组的维度是 (类数, 列数), 当 `transpose = 1` 时, 数组的长度为 (行数, 类数)。其中每一行 (当 `transpose = 0`) 或者每一列 (当 `transpose = 1`) 包含着对应每类对应的数据的平均值。

计算类间距离

根据每个 *items* 的距离函数, 我们可以计算出两个 *clusters* 的距离。两个类别的算术平均值之间的距离通常用于重心法聚类 and *k-means* 聚类, 而 *k-medoids* 聚类中, 通常利用两类的中值进行计算。最短距离法利用的是两类间最近的元素之间的距离, 而最大距离法利用最长的元素之间的距离。在两两平均连锁聚类法中, 类间的距离定义为类内所有对应元素两两间距离的平均值。

为了计算两类之间的距离, 可以利用:

```
>>> from Bio.Cluster import clusterdistance
>>> distance = clusterdistance(data)
```

其中, 包含的参数有：

- `data` (必选) 包含所有元素的矩阵。
- `mask` (默认: `None`) 缺失数据矩阵。若 `mask[i,j]==0`, 则 `data[i,j]` 缺失。若 `mask==None`, 则明没有缺失数据。
- `weight` (默认: `None`) 权重矩阵。若 `weight==None`, 则假设所有的数据使用相同的权重。
- `index1` (默认: `0`) 第一个类所包含的元素索引的列表。如果一个类别只包含一个元素 *i*, 则数据类型可以为一个列表 `[i]`, 或者整数 *i*。
- `index2` (默认: `0`) 第二个类所包含的元素的列表。如果一个类别只包含一个元素 *i*, 则数据类型可以为一个列表 `[i]`, 或者整数 *i*。
- `method` (默认: `'a'`) 选择计算类别间距离的方法:
 - `'a'`: 使用两个类中心的距离 (算术平均值);
 - `'m'`: 使用两个类中心的距离 (中值);

- 's': 使用两类中最短的两个元素之间的距离;
- 'x': 使用两类中最长的两个元素之间的距离;
- 'v': 使用两类中对应元素间的距离的平均值作为距离。
- `dist` (默认: 'e', Euclidean distance) 选择距离函数 (具体见 15.1).
- `transpose` (默认: 0) 选择使用 data 中的行 (`transpose==0`), 或者列 (`transpose==1`) 来计算距离。

15.3 划分算法

划分算法依据所有元素到各自聚类中心距离之和最小化原则, 将元素分为 k 类。类的个数 k 由用户定义。Bio.Cluster 提供了三种不同的算法:

- k -means 聚类
- k -medians 聚类
- k -medoids 聚类

这些算法的区别在于如何定义聚类中心。在 k -means 中, 聚类中心定义为该类中所有元素的平均值。在 k -medians 聚类中, 利用每个维度的中间值来计算。最后, k -medoids 聚类中, 聚类中心定义为该类中, 距离其他所有元素距离之和最小的元素所在的位置。这个方法适用于已知距离矩阵, 但是原始数据矩阵未知的情况, 例如根据结构相似度对蛋白进行聚类。

expectation-maximization (EM) 算法通常用于将数据分成 k 组。在 EM 算法的起始阶段, 随机的把元素分配到不同的组。为了保证所有的类都包含元素, 可以利用二项分布的方法随机为每类挑选元素。然后, 随机的对分组进行排列, 保证每个元素有相同的概率被分到任何一个类别。最终, 保证每类中至少含有一个元素。

之后进行迭代:

- 利用均值, 中值或者 medoid 计算每类的中心;
- 计算每类的元素离各自中心的距离;
- 对于每个元素, 判别其离哪个聚类中心最近;
- 将元素重新分配到最近的聚类, 当不能进行调整时, 迭代终止。

为了避免迭代中产生空的类别, 在 k -means 和 k -medians 聚类中, 算法始终记录着每类中元素的个数, 并且阻止最后一个元素被分到其他的类别中。对于 k -medoids 聚类, 这种检查就是没有必要的, 因为当只剩最后一个元素时, 它离中心的距离为 0, 所以不会被分配到其他的类别中。

由于起始阶段的每类中的元素分配是随机的, 而通常当 EM 算法执行时, 可能产生不同的聚类结果。为了找到最优的聚类结果, 可以对进行 k -means 算法重复多次, 每次都以不同的随机分配作为起始。每次运行后, 都会保存所有元素距离其中心距离之和, 并且选择总距离最小的运行结果最为最终的结果。

EM 算法运行的次数取决于需要聚类元素的多少。一般而言, 我们可以根据最优解被发现的次数来选择。这个次数会作为划分算法的返回值。如果最优解被多次返回, 那么不太可能存在比这个更优的解。然后, 如果最优解只被发现一次, 那么可能存在着距离更小的解。但是, 如果需要聚类的元素过多的话 (多余几百), 那么很难找到一个全局最优解。

EM 算法会在不能进行任何分配的时候停止。我们注意到, 在某些随机的起始分配中, 由于相同的解会在迭代中周期性的重复, 从而导致 EM 算法的失败。因此, 我们在迭代中也会检查是否有周期性出现的解存在。首先, 在给定数目的迭代后, 当前的聚类结果会保存作为一个参考。之后继续迭代一定次数, 比较该结果同之前保存的结果, 可以确定之前的结果是否重复出现。如果有重复出现, 迭代会终止。如果没有出现, 那么再次迭代后的结果会保存作为新的参考。通常, 会首先重复 10 次迭代, 再保存结果为新的参考。之后, 迭代的次数会翻倍, 保证在长的周期中也可以检测到该解。

k-means and *k*-medians

k-means 和 *k*-medians 算法可以利用 `Bio.Cluster` 中的 `kcluster` 实现:

```
>>> from Bio.Cluster import kcluster
>>> clusterid, error, nfound = kcluster(data)
```

其中，包含的参数有：

- **data** (必选) 包含所有元素的矩阵。
- **nclusters** (默认: 2) 期望的类的数目 *k*。
- **mask** (默认: None) 缺失数据矩阵。若 `mask[i,j]==0`, 则 `data[i,j]` 缺失。若 `mask==None`, 则明没有缺失数据。
- **weight** (默认: None) 权重矩阵。若 `weight==None`, 则假设所有的数据使用相同的权重。
- **transpose** (默认: 0) 选择使用 `data` 中的行 (`transpose==0`), 或者列 (`transpose==1`) 来计算距离。- **npass** (默认: 1) *k*-means/-medians 聚类算法运行的次数，每次运行使用不同的随机的起始值。如果指定了 `initialid`, 程序会忽略 “`npass`” 的值，并且聚类算法只会运行一次。
- **method** (默认: a) 指定聚类中心计算方法:
 - `method=='a'`: 算数平均值 (*k*-means clustering);
 - `method=='m'`: 中值 (*k*-medians clustering).

当指定 `method` 使用其他值时，算法会采用算数平均值。

- **dist** (默认: 'e', Euclidean distance) 选择距离函数 (具体见 15.1)。尽管八种距离都可以用于 `kcluster` 计算，但从经验上来讲，Euclidean 距离适合 *k*-means 算法，city-block 距离适合 *k*-medians。
- **initialid** (默认: None) 指定 EM 算法运行初始的聚类类别。如果 `initialid==None`, 那么每运行一次 EM 算法时，都会采取不同的随机初始聚类，总共运行的次数由 `npass` 决定。如果 `initialid` 不是 None, 那么它应该为一个长度为类别数的 1 维数组，每类中至少含有一个元素。通常当初始分类确定后，EM 算法的结果也就确定了。

这个函数的返回值为一个包含 (`clusterid`, `error`, `nfound`) 的元组，其中 `clusterid` 是一个整型矩阵，为每行或列所在的类。`error` 是最优聚类解中，每类内距离的总和，`nfound` 指的是最优解出现的次数。

k-medoids 聚类

`kmedoids` 函数根据提供的距离矩阵和聚类数，来运行 *k*-medoids 聚类：

```
>>> from Bio.Cluster import kmedoids
>>> clusterid, error, nfound = kmedoids(distance)
```

其中，包含的参数有: `nclusters=2, npass=1, initialid=None` |

- **distance** (必选) 两两元素间的距离矩阵，可以通过三种不同的方法提供：
 - 提供一个 2D 的 Numerical Python 数组 (函数只会使用矩阵里左下角数据):


```
distance = array([[0.0, 1.1, 2.3],
                  [1.1, 0.0, 4.5],
                  [2.3, 4.5, 0.0]])
```
 - 输入一个一维的 Numerical Python 数组，包含了距离矩阵左下角的数据：

```
distance = array([1.1, 2.3, 4.5])
```

– 输入一个列表，包含距离矩阵左下角的数据：

```
distance = [array([],
                  array([1.1]),
                  array([2.3, 4.5])
                ]
```

三种方法对应着同样的距离矩阵。

- **nclusters** (默认: 2) 期望的类的数目 k 。
- **npass** (默认: 1) k -medoids 聚类算法运行的次数，每次运行使用不同的随机的起始值。如果指定了 **initialid**, **npass** 的值会忽略，并且聚类算法只会运行一次。
- **initialid** (默认: None) 指定 EM 算法运行初始的聚类类别。如果 **initialid**==None, 那么每运行一次 EM 算法时，都会采取不同的随机初始聚类，总共运行的次数由 **npass** 决定。如果 **initialid** 不是 None, 那么它应该为一个长度为类别数的 1 维数组，每类中至少含有一个元素。通常当初始分类确定后，EM 算法的结果也就确定了。

函数返回值为一个包含 (**clusterid**, **error**, **nfound**) 的元组, 其中 **clusterid** 一个整型矩阵，为每行或列类所在的类。**error** 是在最优解中，类内距离的总和，**nfound** 指的是最优解出现的次数。需要注意的是，**clusterid** 中的类号是指的是代表聚类中心的元素号。

15.4 系统聚类

系统聚类同 k -means 聚类有本质的不同。在系统聚类中，基因间或者实验条件间的相似度是通过树的形式展现出来的。由于可以利用 Treeview 或者 Java Treeview 来查看这些树的结构，因此系统聚类在基因表达谱数据中得到普遍应用。

系统聚类的第一步是计算所有元素间的距离矩阵。之后，融合两个最近的元素成为一个节点。然后，不断的通过融合相近的元素或者节点来形成新的节点，直到所有的元素都属于同一个节点。在追溯元素和节点融合的过程的同时形成了树的结构。不同于 k -means 使用的 EM 算法，系统聚类的过程是固定的。

系统聚类也存在着几个不同的方法，他们区别在于如何计算子节点间的距离。在 Bio.Cluster 中，提供了最短距离法 (pairwise single)，最长距离法 (maximum)，类平均法 (average)，和重心法 (centroid linkage)。

- 在最短距离法中，节点间的距离被定义两个节点最近样品间距离。
- 在最长距离法中，节点间的距离被定义两个节点最远样品间距离。
- 在类平均法中，节点间的距离被定义为所有样品对之间的平均距离。
- 在重心法中，节点间的距离被定义为两个节点重心间的距离。重心的计算是通过对每类中所有元素进行计算的。由于每次都要计算新的节点与其他元素和已存在节点的距离，因此重心法的运行时间比其他系统聚类的方法更长。该方法另外一个特性是，当聚类树的长大的时候，距离并不会增加，有时候反而减少。这是由于使用 Pearson 相关系数作为距离时，对重心的计算和距离的计算不一致产生: 因为 Pearson 相关系数在计算距离时会对数据进行有效归一化，但是重心的计算不会存在该种归一化。

对于最短距离法，最长距离法和类平均法时，两个节点之间的距离是直接对类别里的元素计算得到的。因此，聚类的算法在得到距离矩阵后，不一定需要提供最开始的基因表达数据。而对于重心法而言，新生成的节点的中心必须依靠原始的数据，而不是仅仅依靠距离矩阵。

最短距离法的实现是根据 SLINK algorithm (R. Sibson, 1973), 这个算法具有快速和高效的特点。并且这个方法聚类的结果同传统的方法结果一致。并且该算法，也可以有效的运用于大量的数据，而传统的算法则需要大量的内存需求和运行时间。

展示系统聚类的结果

系统聚类的结果是用树的结构展示所有节点，每个节点包含两个元素或者子节点。通常，我们既关心那个元素或者哪个子节点互相融合，也关心二者之间的距离（或者相似度）。我们可以调用 `Bio.Cluster` 中的 `Node` 类，来存储聚类树的一个节点。 `Node` 的实例包含以下三个属性：

- `left`
- `right`
- `distance`

其中，`left` 和 `right` 是合并到该节点两个元素或子节点的编号。 `distance` 指的是二者间的距离。其中元素的编号是从 0 到（元素数目 -1），而聚类的组别是从 -1 到 -（元素数目 -1）。请注意，节点的数目比元素的数目少一。

为了创建一个新的 `Node` 对象，我们需要指定 `left` 和 `right`；`distance` 是可选的。

```
>>> from Bio.Cluster import Node
>>> Node(2,3)
(2, 3): 0
>>> Node(2,3,0.91)
(2, 3): 0.91
```

对于已存在 `Node` 对象的 `left`, `right`, 和 `distance` 都是可以直接修改的：

```
>>> node = Node(4,5)
>>> node.left = 6
>>> node.right = 2
>>> node.distance = 0.73
>>> node
(6, 2): 0.73
```

当 `left` 和 `right` 不是整数的时候，或者 `distance` 不能被转化成浮点值，会抛出错误。

Python 的类 `Tree` 包含着整个系统聚类的结果。 `Tree` 的对象可以通过一个 `Node` 的列表创建：

```
>>> 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
```

``Tree`` 的初始器会检查包含节点的列表是否是一个正确的系统聚类树的结果：

```
>>> 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
```

也可以使用中括号来对 `Tree` 对象进行检索：

```
>>> 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
```

因为 Tree 对象是只读的，我们不能对 Tree 对象中任何一个节点进行改变。然而，我们可以将其转换成一个节点的列表，对列表进行操作，最后创建新的树。

```
>>> 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
```

这个性质保证了“Tree”结果的正确性。

为了利用可视化工具，例如 Java Treeview，来查看系统聚类树，最好对所有节点的距离进行标准化，使其位于 0 和 1 之间。可以通过对 Tree 对象调用 scale 方法来实现这个功能：

```
>>> tree.scale()
```

这个方法不需要任何参数，返回值是 None。

经过系统聚类后，可以对 Tree 对象进行剪接，将所有的元素分为 k 类：

```
>>> clusterid = tree.cut(nclusters=1)
```

其中 nclusters (默认是 1) 是期望的类别数 k 。这个方法会忽略树结构里面的最高的 $k-1$ 节点，最终形成 k 个独立的类别。对于 k 必须为正数，并且小于或者等于元素的数目。这个方法会返回一个数组 clusterid，包含着每个元素对应的类。

运行系统聚类

为了进行系统聚类，可以用 Bio.Cluster 中的 treecluster 函数。

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(data)
```

包括以下参数：

- data 包含所有元素的矩阵。
- mask (默认: None) 缺失数据矩阵。若 mask[i,j]==0, 则 data[i,j] 缺失。若 mask==None, 则明没有缺失数据。
- weight (默认: None) 权重矩阵。若 weight==None, 则假设所有的数据使用相同的权重。
- transpose (默认: 0) 选择使用 data 中的行 (transpose==0), 或者列 (transpose==1) 来计算距离。
- method (默认: 'm') 选择节点间距离计算方法:
 - method=='s': 最小距离法
 - method=='m': 最大距离法

- method=='c': 重心法
- method=='a': 类平均法
- dist (默认: 'e', Euclidean distance) 选择距离函数 (具体见 15.1).

为了对距离矩阵进行系统聚类, 可以在调用 `treecluster` 时, 用 `distancematrix` 参数来代替 `data` 参数:

```
>>> from Bio.Cluster import treecluster
>>> tree = treecluster(distancematrix=distance)
```

这种情况下, 需要定义下列参数:

- `distancematrix` 元素两两间的距离矩阵, 可以通过三种不同的方法提供:
 - 提供一个 2D 的 Numerical Python 数组 (函数只会使用矩阵里左下角数据):

```
distance = array([[0.0, 1.1, 2.3],
                  [1.1, 0.0, 4.5],
                  [2.3, 4.5, 0.0]])
```

- 输入一个一维的 Numerical Python 数组, 包含了距离矩阵左下角的数据:

```
distance = array([1.1, 2.3, 4.5])
```

- 输入一个列表, 包含距离矩阵左下角的数据:

```
distance = [array([]),
            array([1.1]),
            array([2.3, 4.5])]
```

三种方法对应着同样的距离矩阵。由于 `treecluster` 会对距离矩阵中的值进行随机洗牌, 如果后面需要调用这个距离矩阵, 请在调用 `treecluster` 之前, 事先存到一个新的变量

- `method` 选择节点间距离计算方法:

- method=='s': 最小距离法
- method=='m': 最大距离法
- method=='a': 类平均法

其中, 最小距离法、最大距离法和类平均法可以只通过距离矩阵计算, 而重心法却不行。

当调用 `treecluster` 时, `data` 或者 `distancematrix` 总有一个必须为 `None`。

函数返回一个 `Tree` 对象, 该对象包含着 (元素数目 - 1) 个节点, 当选择行作为聚类时, 元素的数目同行数一致; 当使用列作为聚类时, 元素的数目同列数一致。每个节点都意味着一对相邻连锁的事件, 其中节点的性质 `left` 和 `right` 包含着每个合并的元素或者子节点的编号, `distance` 是两个合并元素或者子节点的距离。元素编号是从 0 到 (元素数目 - 1), 而类别是从 -1 到 (元素数目 - 1)

15.5 Self-Organizing Maps

Self-Organizing Maps (SOMs) 是由 Kohonen 在描述神经网络的时候发明的 (see for instance Kohonen, 1997 [24]). Tamayo (1999) 第一次讲 Self-Organizing Maps 应用到基因表达数据上。[30].

SOMs 根据某种拓扑结果将元素进行分类。通常选用的是矩形的拓扑结构。在 SOMs 生成的类别中, 相邻的两个类的拓扑结构相似度高于他们对其他的相似度。

计算 SOM 的第一步是随机分配数据向量到每个类别中, 如果使用行进行聚类, 那么每个数据向量中的元素个数等于列数。

一个 SOM 会一次读入一行，并且找到该向量最近的拓扑聚类结构。之后利用找到的数据向量对这个类别的数据向量和相邻的类别的数据向量进行调整。调整如下：

$$\Delta \underline{x}_{\text{cell}} = \tau \cdot (\underline{x}_{\text{row}} - \underline{x}_{\text{cell}}).$$

参数 τ 会随着迭代次数增加而减少。可以用一个简单的线性函数来定义其与迭代次数的关系：

$$\tau = \tau_{\text{init}} \cdot \left(1 - \frac{1}{n}\right),$$

τ_{init} 是指定的起始的 τ 值， i 是当前迭代的次数， n 是总的需要迭代的次数。在迭代开始时， τ 变化很快，然而在迭代末尾，变化越来越小。

所有在半径 R 内的类别都会在每次迭代中进行调整。半径也会随着迭代的增加而减小：

$$R = R_{\text{max}} \cdot \left(1 - \frac{1}{n}\right),$$

其中最大的半径定义为：

$$R_{\text{max}} = \sqrt{N_x^2 + N_y^2},$$

其中 (N_x, N_y) 是定义拓扑结构的矩形维度。

函数 `somcluster` 可以用来在一个矩形的网格中计算 Self-Organizing Map。首先，初始化一个随机数产生器。利用随机化产生器来对节点数据进行初始化。在 SOM 中，基因或者芯片的调整顺序同样是随机的。用户可以定义总的 SOM 迭代的次数。

运行 `somcluster`，例如：

```
>>> from Bio.Cluster import somcluster
>>> clusterid, celldata = somcluster(data)
```

其中，可以定义一下参数：

- **data (required)** 包含所有元素的矩阵。
- **mask** (默认: None) 缺失数据矩阵。若 `mask[i,j]==0`，则 `data[i,j]` 缺失。若 `mask==None`，则明没有缺失数据。
- **weight** (默认: None) 权重矩阵。若 `weight==None`，则假设所有的数据使用相同的权重。
- **transpose** (默认: 0) 选择使用 `data` 中的行 (`transpose==0`)，或者列 (`transpose==1`) 来聚类。
- **nxgrid, nygrid** (默认: 2, 1) 当 Self-Organizing Map 计算的时候，矩形的网格所包含的横向和纵向的格子。
- **inittau** (默认: 0.02) SOM 算法中，参数 τ 的初始值，默认是 0.02。这个初始值同 Michael Eisen's Cluster/TreeView 一致。
- **niter** (默认: 1) 迭代运行的次数。
- **dist** (默认: 'e', Euclidean distance) 选择距离函数 (具体见 15.1)。

这个函数返回的是一个元组 (`clusterid`, `celldata`):

- **clusterid**: 一个两列的数组，行的数目等于待聚类元素的个数。每行包含着在矩形 SOM 网格中，将每个元素分配到的格子的 x 和 y 的坐标。
- **celldata**: 当以行进行聚类时，生成的矩阵维度为 (`nxgrid`, `nygrid`, number of columns)；当以列进行聚类时，生成的矩阵维度为 (`nxgrid`, `nygrid`, number of rows)。在这个矩阵里，`[ix][iy]` 表示着一个一维向量，其中用于计算该类中心的这基因的表达式数据。

15.6 主成分分析

主成分分析 (PCA) 被广泛的用于分析多维数据，一个将主成分分析应用于表达谱数据的请见 Yeung and Ruzzo (2001) [33].

简而言之，PCA 是一种坐标转换的方法，转换后的基础向量成为主成分，变换前的每行可以用主成分的线性关系显示。主成分的选择是基于残差尽可能的小的原则。例如，一个 n 的数据矩阵可以表示为三维空间内的一个椭圆球形的点的云。第一主成分是这个椭圆球形的最长轴，第二主成分是次长轴，第三主成分是最短的轴。矩阵中，每一行都可以用主成分的线性关系展示。一般而言，为了对数据进行降维，只保留最重要的几个主成分。剩余的残差认为是不可解释的方差。

可以通过计算数据的协方差矩阵的特征向量来得到主成分。每个主成分对应的特征值决定了其在数据中代表的方差的大小。

在进行主成分分析前，矩阵的数据每一列都要减去其平均值。在上面椭圆球形云的例子中，数据在 3D 空间中，围绕着其中心分布，而主成分则显示着每个点对其中心的变化。

函数 `pca` 首先使用奇异值分解 (singular value decomposition) 来计算矩阵的特征值和特征向量。奇异值分解使用的是 Algol 写的 C 语言的 `svd` [16]，利用的是 Householder bidiagonalization 和 QR 算法的变异。主成分，每个数据在主成分上的坐标和主成分对应的特征值都会被计算出来，并按照特征值的降序排列。如果需要数据中心，则需要在调用 `pca` 前，对每列数据减去其平均值。

将主成分分析应用于二维矩阵 `data`, 可以：

```
>>> from Bio.Cluster import pca
>>> columnmean, coordinates, components, eigenvalues = pca(data)
```

函数会返回一个元组 `columnmean, coordinates, components, eigenvalues`：

- `columnmean` 包含 `data` 每列均值的数组。
- `coordinates` `data` 中每行数据在主成分上对应的坐标。
- `components` 主成分
- `eigenvalues` 每个主成分对应的特征值

原始的数据 `data` 可以通过计算 `columnmean + dot(coordinates, components)` 得到。

15.7 处理 Cluster/TreeView-type 文件

Cluster/TreeView 是一个对基因表达数据可视化的工具。他们最初由 Michael Eisen 在 Stanford University 完成。Bio.Cluster 包含着读写 Cluster/TreeView 对应的文件格式的函数。因此，将结果保存为该格式后，可以用 Treeview 对结果进行直接的查看。我们推荐使用 Alok Saldanha 的 <http://jtreeview.sourceforge.net/> Java TreeView 程序。这个软件可以显示系统聚类和 k -means 聚类的结果。

类 `Record` 的一个对象包含着一个 Cluster/TreeView-type 数据文件需要的所有信息。为了将结果保存到一个 `Record` 对象中，首先需要打开一个文件，并读取：

```
>>> from Bio import Cluster
>>> handle = open("mydatafile.txt")
>>> record = Cluster.read(handle)
>>> handle.close()
```

两步操作使得你可以较灵活地操作不同来源的数据，例如：

```
>>> import gzip # Python standard library
>>> handle = gzip.open("mydatafile.txt.gz")
```

来打开一个 gzipped 文件，或者利用

```
>>> import urllib # Python standard library
>>> handle = urllib.urlopen("http://somewhere.org/mydatafile.txt")
```

来打开一个网络文件，然后调用 read.

read 命令会读取一个由制表符分割的文本文件 mydatafile.txt，文件包含着符合 Michael Eisen's Cluster/TreeView 格式的基因表达数据。具体的格式说明，可以参见 Cluster/TreeView 手册，链接见 [Michael Eisen's lab website](#) 或者 [our website](#).

一个 Record 对象有以下的性质:

- **data** 包含基因表达数据的矩阵，每行为基因，每列为芯片。
- **mask** 缺失值的整型数组。如果 mask[i,j]==0, 则 data[i,j] 是缺失的. 如果 mask==None, 那么没有数据缺失。
- **geneid** 包含每个基因的独特说明的列表 (例如 ORF 数目).
- **genename** 包含每个基因说明的列表 (例如基因名)。如果文件中不包含该数据，那么 genename 被设为 None.
- **gweight** 计算表达谱数据中，基因间的距离使用的权重。如果文件中不含该信息，则 gweight 为 None.
- **gorder** 期望输出文件中基因的排列的顺序。如果文件中不含该信息，则 gorder 为 “None”。
- **expid** 包含每个芯片说明的列表，例如实验条件。
- **eweight** 计算表达谱数据中，不同芯片间的距离使用的权重。如果文件中不含该信息，则 eweight 为 None.
- **eorder** 期望输出文件中基因的排列的顺序。如果文件中不含该信息，则 eorder 为 None.
- **uniqid** 用于代替文件中 UNIQID 的字符串。

在载入 Record 对象后，上述的每个性质可以直接读取和修改。例如，可以对 record.data 直接取对数来对数据进行 log 转换。

计算距离矩阵

为了计算 record 中存储元素的距离矩阵，可以用：

```
>>> matrix = record.distancematrix()
```

其中，包含以下参数：

- **transpose** (默认: 0) 选择对 data 的行 (transpose==0), 或者列 (transpose==1) 计算距离。
- **dist** (默认: 'e', Euclidean distance) 选择合适的元素距离算法 (见 15.1).

函数会返回一个距离矩阵，每行的列数等于行数。(见 15.1).

计算聚类中心

为了计算存储在 record 中的元素的聚类中心，利用：

```
>>> cdata, cmask = record.clustercentroids()
```

- **clusterid** (默认: None) 展示每个元素所属类的整型向量。如果缺少 clusterid, 默认所有的元素属于同一类。

- `method` (默认: 'a') 选择使用算术平均值 (`method=='a'`) 或者中值 (`method=='m'`) 来计算聚类中心。
 - `transpose` (默认: 0) 选择计算 “data” 的行 (`transpose==0`), 或者列 (`transpose==1`) 计算中心。
- 函数返回元组 `cdata`, `cmask`; 见 [15.2](#).

计算两类间的距离

为了计算存储在 `record` 中的两类的距离, 利用:

```
>>> distance = record.clusterdistance()
```

其中, 包含以下参数:

- `index1` (默认: 0) 第一个类别所包含的元素的列表。如果一个类别只包含一个元素 *i* 可以为一个列表 `[i]`, 或者整数 *i*.
- `index2` (默认: 0) 第二个类别所包含的元素的列表。如果一个类别只包含一个元素 *i* 可以为一个列表 `[i]`, 或者整数 *i*.
- `method` (默认: 'a') 选择计算类别间距离的方法:
 - 'a': 使用两个聚类中心的距离 (算术平均值);
 - 'm': 使用两个聚类中心的距离 (中值);
 - 's': 使用两类中最短的两个元素之间的距离;
 - 'x': 使用两类中最长的两个元素之间的距离;
 - 'v': 使用两类中两两元素距离的平均值作为距离。
- `dist` (默认: 'e', Euclidean distance) 选择使用的距离函数 (见 [15.1](#)).
- `transpose` (默认: 0) 选择使用 `data` 的行 (`transpose==0`), 或者列 (`transpose==1`) 计算距离。

进行系统聚类

为了对存储在 `record` 中的数据进行系统聚类, 利用:

```
>>> tree = record.treecluster()
```

包含以下参数:

- `transpose` (默认: 0) 选择使用行 (`transpose==0`) 或者列 (`transpose==1`) 用于聚类
- `method` (默认: 'm') 选择合适的节点距离计算方法:
 - `method=='s'`: 最小距离法
 - `method=='m'`: 最大距离法
 - `method=='c'`: 重心法
 - `method=='a'`: 类平均法
- `dist` (默认: 'e', Euclidean distance) 选择使用的距离函数 (见 [15.1](#)).
- `transpose` 选择使用基因或者芯片进行聚类, 如果是 `transpose==0`, 则使用基因 (行) 进行聚类, 如果使用 `transpose==1`, 芯片 (列) 用于聚类。

函数返回 Tree 对象。对象包含 (元素数目-1) 节点, 如果使用行进行聚类时, 元素数目为总行数; 当使用列进行聚类时, 元素数目为总列数。每个节点描述着一对节点连接, 然而节点的性质 left 和 right 包含着相邻节点所有的元素和子节点数, distance 显示着左右节点的距离。元素从 0 到 (元素数目-1) 进行索引, 而类别从 -1 to 0 (元素数目-1) 进行索引。

进行 *k*-means or *k*-medians 聚类

为了对存储在 record 中的元素进行 *k*-means 或者 *k*-medians 聚类, 可以使用:

```
>>> clusterid, error, nfound = record.kcluster()
```

包含以下参数:

- **nclusters** (默认: 2) 类的数目 *k*.
- **transpose** (默认: 0) 选择使用 data 的行 (transpose==0), 或者列 (transpose==1) 计算距离。
- **npass** (默认: 1) *k*-means/-medians 聚类算法运行的次数, 每次运行使用不同的随机的起始值。如果指定了 initialid, npass 的值会忽略, 并且聚类算法只会运行一次。
- **method** (默认: a) 指定确定聚类中心的方法:
 - method=='a': 算数平均值 (*k*-means clustering);
 - method=='m': 中间值 (*k*-medians clustering).

当指定 method 使用其他值时, 算法会采用算数平均值。

- **dist** (默认: 'e', Euclidean distance) 选择使用的距离函数 (见 15.1).

这个函数返回的是一个元组 (clusterid, error, nfound), 其中 clusterid 是一个每行或则列对应的类的编号。error 是最优解的类内的距离和, nfound 是最优解被发现的次数。

计算 Self-Organizing Map

可以利用以下命令, 计算对存储在 record 中元素计算 Self-Organizing Map :

```
>>> clusterid, celldata = record.somcluster()
```

包含以下参数:

- **transpose** (默认: 0) 选择使用 data 的行 (transpose==0), 或者列 (transpose==1) 计算距离。
- **nxgrid, nygrid** (默认: 2, 1) 当 Self-Organizing Map 计算时, 在矩形网格里的横向和纵向格子数目
- **inittau** (默认: 0.02) 用于 SOM 算法的参数 τ 的初始值。默认的是 inittau 是 0.02, 同 Michael Eisen's Cluster/TreeView 程序中使用的参数一致。
- **niter** (默认: 1) 迭代运行的次数。
- **dist** (默认: 'e', Euclidean distance) 选择使用的距离函数 (见 15.1).

函数返回一个元组 (clusterid, celldata):

- **clusterid**: 一个二维数组, 行数同待聚类的元素数目相同。每行的内容对应着该元素在矩形 SOM 格子内 *x* 和 *y* 的坐标。
- **celldata**: 格式为一个矩阵, 如果是对行聚类, 内容为 (nxgrid, nygrid, 列数), 如果是对列聚类, 那么内容为 (nxgrid, nygrid, 行数)。矩阵中, 坐标 [ix][iy] 对应的是该坐标的网格里的基因表达数据的聚类中心的一维向量。

保存聚类结果

为了保存聚类结果，可以利用：

```
>>> record.save(jobname, geneclusters, expclusters)
```

包含以下参数：

- `jobname` 字符串 `jobname` 作为保存的文件名。
- `geneclusters` 这个参数指的是基因（以行聚类）的结果。在 *k*-means 聚类中，这个参数是一个一维的数组，包含着每个基因对应的类别，可以通过 `kcluster` 得到。在系统聚类中，`geneclusters` 是一个 `Tree` 对象。
- `expclusters` 这个参数指的是实验条件（以列聚类）的结果。在 *k*-means 聚类中，这个参数是一个一维的数组，包含着每个实验条件对应的类别，可以通过 `kcluster` 得到。在系统聚类中，`geneclusters` 是一个 “`Tree`” 对象。

这个方法会生成文本文件 `jobname.cdt`, `jobname.gtr`, `jobname.atr`, `jobname*.kgg`, 和/或 `jobname*.kag`。这些文件可以用于后续分析。如果 `geneclusters` 和 `expclusters` 都是 `None`，那这个方法只会生成 `jobname.cdt`；这个文件可以被读取，生成一个新的 `Record` 对象。

15.8 示例

以下是一个系统聚类的例子，其中使用最短距离法对基因进行聚类，用最大距离法对实验条件进行聚类。由于使用 Euclidean 距离对基因进行聚类，因此需要将节点距离 `genetree` 进行调整，使其处于 0 和 1 之间。这种调整对于 Java `TreeView` 正确显示树结构也是很必须的。同时使用 `uncentered correlation` 对实验条件进行聚类。在这种情况下，不需要任何的调整，因为 `exptree` 中的结果已经位于 0 和 2 之间。示例中使用的文件 `cyano.txt` 可以从 `data` 文件夹中找到。

```
>>> from Bio import Cluster
>>> handle = open("cyano.txt")
>>> record = Cluster.read(handle)
>>> handle.close()
>>> genetree = record.treecluster(method='s')
>>> genetree.scale()
>>> exptree = record.treecluster(dist='u', transpose=1)
>>> record.save("cyano_result", genetree, exptree)
```

这个命令会生成 `cyano_result.cdt`, `cyano_result.gtr`, 和 `cyano_result.atr` 等文件。

同样的，也可以保存一个 *k*-means 聚类的结果：

```
>>> from Bio import Cluster
>>> handle = open("cyano.txt")
>>> record = Cluster.read(handle)
>>> handle.close()
>>> (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)
```

上述代码将生成文件 `cyano_result_K_G2_A2.cdt`, `cyano_result_K_G2.kgg`, 和 `cyano_result_K_A2.kag`。

15.9 附加函数

`median(data)` 返回一维数组 `data` 的中值

`mean(data)` 返回一维数组 `data` 的均值。

`version()` 返回使用的 C 聚类库的版本号。

第 16 章监督学习方法

注意本章介绍的所有监督学习方法都需要先安装 Numerical Python (numpy)。

16.1 Logistic 回归模型

16.1.1 背景和目的

Logistic 回归是一种监督学习方法，通过若干预测变量 x_{i*} 的加权和来尝试将样本划分为 K 个不同类别。Logistic 回归模型可用来计算预测变量的权重 β_{i*} 。在 Biopython 中，logistic 回归模型目前只实现了二类别 ($K = 2$) 分类，而预测变量的数量没有限制。

作为一个例子，我们试着预测细菌中的操纵子结构。一个操纵子是在一条 DNA 链上许多相邻基因组成的一个集合，可以被共同转录为一条 mRNA 分子。这条 mRNA 分子经翻译后产生多个不同的蛋白质。我们将以枯草芽孢杆菌的操纵子数据进行说明，它的一个操纵子平均包含 2.4 个基因。

作为理解细菌的基因调节的第一步，我们需要知道其操纵子的结构。枯草芽孢杆菌大约 10% 的基因操纵子结构已经通过实验获知。剩下的 90% 的基因操纵子结构可以通过一种监督学习方法来预测。

在这种监督学习方法中，我们需要选择某些与操纵子结构有关的容易度量的预测变量 x_{i*} 。例如可以选择基因间碱基对距离来作为其中一个预测变量。同一个操纵子中的相邻基因往往距离相对较近，而位于不同操纵子的相邻基因间通常具有更大的空间来容纳启动子和终止子序列。另一个预测变量可以基于基因表达量度。根据操纵子的定义，属于同一个操纵子的基因有相同的基因表达谱，而不同操纵子的两个基因的表达谱也不相同。在实际操作中，由于存在测量误差，对相同操纵子的基因表达轮廓的测量不会完全一致。为了测量基因表达轮廓的相似性，我们假设测量误差服从正态分布，然后计算对应的对数似然分值。

现在我们有了两个预测变量，可以据此预测在同一条 DNA 链上两个相邻基因是否属于相同的操纵子：
- x_1 ：两基因间的碱基对数；
- x_2 ：两基因表达谱的相似度。

在 logistic 回归模型中，我们使用这两个预测变量的加权和来计算一个联合得分 S ：

$$S = \beta_0 + \beta_1 x_1 + \beta_2 x_2. (16.0)$$

根据下面两组示例基因，logistic 回归模型对参数 β_0 ， β_1 ， β_2 给出合适的值：
- OP: 相邻基因，相同 DNA 链，属于相同操纵子；
- NOP: 相邻基因，相同 DNA 链，属于不同操纵子。

在 logistic 回归模型中，属于某个类别的概率依赖于通过 logistic 函数得出的分数。对于这两类 OP 和 NOP，相应概率可如下表述：

to

$$\begin{aligned} \Pr(\text{OP}|x_1, x_2) &= \frac{\exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \\ \Pr(\text{NOP}|x_1, x_2) &= \frac{1}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \quad (16.0) \\ &= \frac{\exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \Pr(\text{NOP}|x_1, x_2) \\ &= \frac{1}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \quad (16.1) \end{aligned}$$

使用一组已知是否属于相同操纵子（OP 类别）或不同操纵子（NOP 类别）的基因对，通过最大化相应概率函数的对数似然值，我们可以计算权重 $\beta_0, \beta_1, \beta_2$ 。(16.2) 和 (16.3)。

16.1.2 训练 logistic 回归模型

表 16.1：已知类别 (OP or NOP) 的相邻基因对. 如果两个基因相重叠，其基因间距离为负值

基因对	基因间距离 (x_1)	基因表达得分 (x_2)	类别
<i>cotJA</i> — <i>cotJB</i>	-53	-200.78	OP
<i>yesK</i> — <i>yesL</i>	117	-267.14	OP
<i>lplA</i> — <i>lplB</i>	57	-163.47	OP
<i>lplB</i> — <i>lplC</i>	16	-190.30	OP
<i>lplC</i> — <i>lplD</i>	11	-220.94	OP
<i>lplD</i> — <i>yetF</i>	85	-193.94	OP
<i>yfmT</i> — <i>yfmS</i>	16	-182.71	OP
<i>yfmF</i> — <i>yfmE</i>	15	-180.41	OP
<i>citS</i> — <i>citT</i>	-26	-181.73	OP
<i>citM</i> — <i>yflN</i>	58	-259.87	OP
<i>yfiI</i> — <i>yfiJ</i>	126	-414.53	NOP
<i>lipB</i> — <i>yfiQ</i>	191	-249.57	NOP
<i>yfiU</i> — <i>yfiV</i>	113	-265.28	NOP
<i>yfhH</i> — <i>yfhI</i>	145	-312.99	NOP
<i>cotY</i> — <i>cotX</i>	154	-213.83	NOP
<i>yjoB</i> — <i>rapA</i>	147	-380.85	NOP
<i>ptsI</i> — <i>splA</i>	93	-291.13	NOP

表 16.1 列出了枯草芽孢杆菌的一些基因对，这些基因的操纵子结构已知。让我们根据表中的这些数据来计算其 logistic 回归模型：

```
>>> from Bio import LogisticRegression
>>> xs = [[-53, -200.78],
          [117, -267.14],
          [57, -163.47],
          [16, -190.30],
          [11, -220.94],
          [85, -193.94],
          [16, -182.71],
          [15, -180.41],
          [-26, -181.73],
          [58, -259.87],
          [126, -414.53],
          [191, -249.57],
          [113, -265.28],
          [145, -312.99],
          [154, -213.83],
          [147, -380.85],
          [93, -291.13]]
>>> ys = [1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          0,
          0,
          0,
          0,
          0,
          0,
          0]
>>> model = LogisticRegression.train(xs, ys)
```

这里，`xs` 和 `ys` 是训练数据：`xs` 包含每个基因对的预测变量，`ys` 指定是否这个基因对属于相同操纵子（1，类别 OP）或不同操纵子（0，类别 NOP）。Logistic 回归模型结果存储在 `model` 中，包含权重 β_0 , β_1 , and β_2 ：

```
>>> model.beta
[8.9830290157144681, -0.035968960444850887, 0.02181395662983519]
```

注意 β_1 是负的，这是因为具有更短基因间距离的基因对有更高的概率属于相同操纵子（类别 OP）。另一方面， β_2 为正，因为属于相同操纵子的基因对通常有更高的基因表达谱相似性得分。参数 β_0 是正值是因为在这个训练数据中操纵子基因对占据大多数。

函数 `train` 有两个可选参数：`update_fn` 和 `typecode`。`update_fn` 可用来指定一个回调函数，以迭代数和対数似然值做参数。在这个例子中，我们可以使用这个回调函数追踪模型计算（使用 Newton-Raphson 迭代来最大化 logistic 回归模型的对数似然函数）进度：

```
>>> def show_progress(iteration, loglikelihood):
    print "Iteration:", iteration, "Log-likelihood function:", loglikelihood
```

```
>>>
>>> model = LogisticRegression.train(xs, ys, update_fn=show_progress)
Iteration: 0 Log-likelihood function: -11.7835020695
Iteration: 1 Log-likelihood function: -7.15886767672
Iteration: 2 Log-likelihood function: -5.76877209868
Iteration: 3 Log-likelihood function: -5.11362294338
Iteration: 4 Log-likelihood function: -4.74870642433
Iteration: 5 Log-likelihood function: -4.50026077146
Iteration: 6 Log-likelihood function: -4.31127773737
Iteration: 7 Log-likelihood function: -4.16015043396
Iteration: 8 Log-likelihood function: -4.03561719785
Iteration: 9 Log-likelihood function: -3.93073282192
Iteration: 10 Log-likelihood function: -3.84087660929
Iteration: 11 Log-likelihood function: -3.76282560605
Iteration: 12 Log-likelihood function: -3.69425027154
Iteration: 13 Log-likelihood function: -3.6334178602
Iteration: 14 Log-likelihood function: -3.57900855837
Iteration: 15 Log-likelihood function: -3.52999671386
Iteration: 16 Log-likelihood function: -3.48557145163
Iteration: 17 Log-likelihood function: -3.44508206139
Iteration: 18 Log-likelihood function: -3.40799948447
Iteration: 19 Log-likelihood function: -3.3738885624
Iteration: 20 Log-likelihood function: -3.3423876581
Iteration: 21 Log-likelihood function: -3.31319343769
Iteration: 22 Log-likelihood function: -3.2860493346
Iteration: 23 Log-likelihood function: -3.2607366863
Iteration: 24 Log-likelihood function: -3.23706784091
Iteration: 25 Log-likelihood function: -3.21488073614
Iteration: 26 Log-likelihood function: -3.19403459259
Iteration: 27 Log-likelihood function: -3.17440646052
Iteration: 28 Log-likelihood function: -3.15588842703
Iteration: 29 Log-likelihood function: -3.13838533947
Iteration: 30 Log-likelihood function: -3.12181293595
Iteration: 31 Log-likelihood function: -3.10609629966
Iteration: 32 Log-likelihood function: -3.09116857282
Iteration: 33 Log-likelihood function: -3.07696988017
Iteration: 34 Log-likelihood function: -3.06344642288
Iteration: 35 Log-likelihood function: -3.05054971191
Iteration: 36 Log-likelihood function: -3.03823591619
Iteration: 37 Log-likelihood function: -3.02646530573
Iteration: 38 Log-likelihood function: -3.01520177394
Iteration: 39 Log-likelihood function: -3.00441242601
Iteration: 40 Log-likelihood function: -2.99406722296
Iteration: 41 Log-likelihood function: -2.98413867259
```

一旦对数似然函数得分增加值小于 0.01，迭代将终止。如果在 500 次迭代后还没有到达收敛，`train` 函数返回并抛出一个 `AssertionError`。

可选的关键字 `typecode` 几乎可以一直忽略。这个关键字允许用户选择要使用的数值矩阵类型。当为了避免大数据计算的内存问题时，可能有必要使用单精度浮点数（`Float8`，`Float16` 等等）而不是默认的 `double` 型。

16.1.3 使用 logistic 回归模型进行分类

调用 `classify` 函数可以进行分类。给定一个 logistic 回归模型和 x_1 和 x_2 的值（例如，操纵子结构未知的基因对），`classify` 函数返回 1 或 0，分别对应类别 OP 和 NOP。例如，考虑基因对 *ycx*E，*ycx*D 和 *yci*B，*yci*A：

表 16.2 : 操纵子状态未知的相邻基因对

基因对	基因间距离 x_1	基因表达得分 x_2
<i>ycx</i> E — <i>ycx</i> D	6	-173.143442352
<i>yxi</i> B — <i>yxi</i> A	309	-271.005880394

Logistic 回归模型预测 *ycx*E , *ycx*D 属于相同操纵子 (类别 OP), 而 *yxi*B , *yxi*A 属于不同操纵子:

```
>>> print "ycxE, yxcD:", LogisticRegression.classify(model, [6,-173.143442352])
ycxE, yxcD: 1
>>> print "yxiB, yxiA:", LogisticRegression.classify(model, [309, -271.005880394])
yxiB, yxiA: 0
```

(这个结果和生物学文献报道的一致)。

为了确定这个预测的可信度, 我们可以调用 `calculate` 函数来获得类别 OP 和 NOP 的概率 (公式 (16.2) 和 (16.3))。对于 *ycx*E, *ycx*D 我们发现

```
>>> q, p = LogisticRegression.calculate(model, [6,-173.143442352])
>>> print "class OP: probability =", p, "class NOP: probability =", q
class OP: probability = 0.993242163503 class NOP: probability = 0.00675783649744
```

对于 *yxi*B , *yxi*A

```
>>> q, p = LogisticRegression.calculate(model, [309, -271.005880394])
>>> print "class OP: probability =", p, "class NOP: probability =", q
class OP: probability = 0.000321211251817 class NOP: probability = 0.999678788748
```

为了确定回归模型的预测精确性, 我们可以把模型应用到训练数据上:

```
>>> for i in range(len(ys)):
    print "True:", ys[i], "Predicted:", LogisticRegression.classify(model, xs[i])
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
```

这表示除一个基因对外其他所有预测都是正确的。Leave-one-out 分析可以对预测精确性给出一个更可信的估计。Leave-one-out 是指从训练数据中移除要预测的基因重新计算模型, 再用该模型进行预测比对:

```
>>> for i in range(len(ys)):
    model = LogisticRegression.train(xs[:i]+xs[i+1:], ys[:i]+ys[i+1:])
    print "True:", ys[i], "Predicted:", LogisticRegression.classify(model, xs[i])
```

```
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
```

Leave-one-out 分析显示这个 logistic 回归模型的预测只对两个基因对不正确，对应预测精确度为 88%。

16.1.4 Logistic 回归，线性判别分析和支持向量机

Logistic 回归模型类似于线性判别分析。在线性判别分析中，类别概率同样可由方程 (16.2) 和 (16.3) 给出。但是，不是直接估计系数 β ，我们首先对预测变量 x 拟合一个正态分布。然后通过这个正态分布的平均值和方差计算系数 β 。如果 x 的分布确实是正态的，线性判别分析将比 logistic 回归模型有更好的性能。另一方面，logistic 回归模型对于偏态到正态的广泛分布更加强健。

另一个相似的方法是应用线性核函数的支持向量机。这样的 SVM 也使用一个预测变量的线性组合，但是是从靠近类别之间的边界区域的预测变量 x 来估计系数 β 。如果 logistic 回归模型 (公式 (16.2) 和 (16.3)) 很好的描述了远离边界区域的 x ，我们可以期望 logistic 回归模型优于线性核函数 SVM，因为它应用了更多数据。如果不是，SVM 可能更好。

Trevor Hastie, Robert Tibshirani, and Jerome Friedman: The Elements of Statistical Learning. Data Mining, Inference, and Prediction.(统计学习基础: 数据挖掘、推理与预测) Springer Series in Statistics, 2001. 4.4 章.

16.2 k -最近邻居法 (KNN)

16.2.1 背景和目的

最近邻居法是一种不需要将数据拟合到一个模型的监督学习算法。数据点是基于训练数据集的 k 个最近邻居类别进行分类的。

在 Biopython 中， KNN 方法可在 Bio.KNN 中获得。我们使用 16.1 同样的操纵子数据集来说明 Biopython 中 KNN 方法的用法。

16.2.2 初始化一个 KNN 模型

使用表 16.1 中的数据，我们创建和初始化一个 *KNN* 模型：

```
>>> from Bio import kNN
>>> k = 3
>>> model = kNN.train(xs, ys, k)
```


这里 `xs` 和 `ys` 和 16.1.2 中的相同。 k 是分类中的邻居数 k 。对于二分类，为 k 选择一个奇数可以避免 tied votes。函数名 `train` 在这里有点不合适，因为就没有训练模型：这个函数仅仅是用来存储模型变量 `xs`，`ys` 和 k 。

16.2.3 使用 *KNN* 模型来分类

应用 KNN 模型对新数据进行分类，我们使用 `classify` 函数。这个函数以一个数据点 (x_1, x_2) 为参数并在训练数据集 `xs` 中寻找 k -最近邻居。然后基于在这 k 个邻居中出现最多的类别 (`ys`) 来对数据点 (x_1, x_2) 进行分类。

对于基因对 `yxcE`、`yxcD` 和 `yxiB`、`yxiA` 的例子，我们发现：

```
>>> x = [6, -173.143442352]
>>> print "yxcE, yxcD:", kNN.classify(model, x)
yxcE, yxcD: 1
>>> x = [309, -271.005880394]
>>> print "yxiB, yxiA:", kNN.classify(model, x)
yxiB, yxiA: 0
```

和 logistic 回归模型一致，`yxcE,*yxcD*` 被归为一类 (类别 OP)，`yxiB,*yxiA*` 属于不同操纵子。

函数 `classify` 可以指定距离函数和权重函数作为可选参数。距离函数影响作为最近邻居的 k 个类别的选择，因为这些到查询点 (x, y) 有最小距离的类别被定义为邻居。默认使用欧几里德距离。另外，我们也可以如示例中的使用曼哈顿距离：

```
>>> def cityblock(x1, x2):
...     assert len(x1)==2
...     assert len(x2)==2
...     distance = abs(x1[0]-x2[0]) + abs(x1[1]-x2[1])
...     return distance
...
>>> x = [6, -173.143442352]
>>> print "yxcE, yxcD:", kNN.classify(model, x, distance_fn = cityblock)
yxcE, yxcD: 1
```

权重函数可以用于权重投票。例如，相比于相邻较远的邻居，我们可能想给更近的邻居一个更高的权重：

```
>>> def weight(x1, x2):
...     assert len(x1)==2
...     assert len(x2)==2
...     return exp(-abs(x1[0]-x2[0]) - abs(x1[1]-x2[1]))
...
>>> x = [6, -173.143442352]
>>> print "yxcE, yxcD:", kNN.classify(model, x, weight_fn = weight)
yxcE, yxcD: 1
```

默认所有邻居有相同权重。

为了确定这些预测的置信度，我们可以调用函数 `calculate` 来计算分配到类别 OP 和 NOP 的总权重。对于默认的加权方案，这样减少了每个分类的邻居数量。对于 `yxcE`，`yxcD`，我们发现

```
>>> x = [6, -173.143442352]
>>> weight = kNN.calculate(model, x)
>>> print "class OP: weight =", weight[0], "class NOP: weight =", weight[1]
class OP: weight = 0.0 class NOP: weight = 3.0
```

这意味着 `x1`，`x2` 的所有三个邻居都属于 NOP 类别。对另一个例子 `yesK`，`yesL` 我们发现

```
>>> x = [117, -267.14]
>>> weight = kNN.calculate(model, x)
>>> print "class OP: weight =", weight[0], "class NOP: weight =", weight[1]
class OP: weight = 2.0 class NOP: weight = 1.0
```

这意思是两个邻居是操纵子对，另一个是非操纵子对

对于 *KNN* 方法的预测精确性，我们对训练数据应用模型：

```
>>> for i in range(len(ys)):
    print "True:", ys[i], "Predicted:", kNN.classify(model, xs[i])
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
```

显示除了两个基因对所有预测都是正确的。Leave-one-out 分析可以对预测精确性给出一个更可信的估计，这是通过从训练数据中移除要预测的基因，再重新计算模型实现：

```
>>> for i in range(len(ys)):
    model = kNN.train(xs[:i]+xs[i+1:], ys[:i]+ys[i+1:])
    print "True:", ys[i], "Predicted:", kNN.classify(model, xs[i])
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
```

Leave-one-out 分析显示这个 KNN 模型的预测正确 17 个基因对中的 13 个，对应预测精确度为 76%。

16.3 Naive 贝叶斯

这部分将描述模块 `Bio.NaiveBayes` .

16.4 最大熵

这部分将描述模块 `Bio.MaximumEntropy`.

16.5 马尔科夫模型

这部分将描述模块 `Bio.MarkovModel` 和/或 `Bio.HMM.MarkovModel` .

第 17 章 GRAPHICS 模块中的基因组可视化包—GENOMEDIAGRAM

Bio.Graphics 模块基于 Python 的第三方扩展库 [ReportLab](#)，ReportLab 主要生成 PDF 文件，同时也能生成 EPS (Encapsulated Postscript) 文件和 SVG 文件。ReportLa 可以导出矢量图，如果安装依赖关系 (Dependencies)，比如 [PIL\(Python Imaging Library\)](#)，ReportLab 也可以导出 JPEG, PNG, GIF, BMP 和 PICT 格式的位图 (Bitmap image)。

17.1 基因组可视化包—GenomeDiagram

17.1.1 GenomeDiagram 简介

Bio.Graphics.GenomeDiagram 包被整合到 Biopython 1.50 版之前，就已经是 Biopython 的独立模块。GenomeDiagram 包首次出现在 2006 年 Pritchard 等人在 *Bioinformatics* 杂志的一篇文章 [2]，文中展示了一些图像示例，更多图像示例请查看 GenomeDiagram 手册 <http://biopython.org/DIST/docs/GenomeDiagram/userguide.pdf>。正如“GenomeDiagram”名称所指，它主要用于可视化全基因组 (特别是原核生物基因组)，即可绘制线型图也可绘制环形图，Toth 等人在 2006 年发表的文章 [3] 中图 2 就是一个示例。Van der Auwera 等人在 2009 年发表的文章 [4] 中图 1 和图 2 也进一步说明，GenomeDiagram 适用于噬菌体、质粒和线粒体等微小基因组的可视化。

如果存储基因组信息的是从 GenBank 文件中下载的 SeqRecord 话，它会包含许多 SeqFeature，那么用这个模块处理就很简单 (详见第 4 章和第 5 章)。

17.1.2 图形，轨迹，特征集和特征

GenomeDiagram 使用一组嵌套的对象，图层中沿着水平轴或圆圈的图形对象 (diagram object) 表示一个序列 (sequence) 或序列区域 (sequence region)。一个图形可以包含多个轨迹 (track)，呈现为横向排列或者环形放射图。这些轨迹的长度通常相等，代表相同的序列区域。可用一个轨迹表示基因的位置，另一个轨迹表示调节区域，第三个轨迹表示 GC 含量。可将最常用轨迹的特征打包为一个特征集 (feature-sets)。CDS 的特征可以用一个特征集，而 tRNA 的特征可以用另外一个特征集。这不是强制性的要求，你可以在 diagram 中用同样的特征集。如果 diagram 中用不同的特征集，修改一个特征会很容易，比如把所有 tRNA 的特征都变为红色，你只需选择 tRNA 的特征就行。

新建图形主要有两种方式。第一种是自上而下的方法 (Top-Down)，首先新建 diagram 对象，然后用 diagram 的方法来添加 track(s)，最后用 track 的方法添加特征。第二种是自下而上的方法 (Bottom-Up)，首先单独新建对象，然后再将其进行组合。

17.1.3 自上而下的实例

我们用一个从 GenBank 文件中读取出来的 SeqRecord 来绘制全基因组 (详见第 5 章)。这里用鼠疫杆菌 *Yersinia pestis* biovar *Microtus* 的 pPCP1 质粒，元数据文件 NC_005816.gb 在 Biopython 中 GenBank

的 tests 目录下，NC_005816.gb 也可下载

```
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")
```

这里用自上而下的方法，导入目标序列后，新建一个 diagram，然后新建一个 track，最后新建一个特征集（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()
```

接下来的部分最有趣，提取 SeqRecord 中每个基因的 SeqFeature 对象，就会为 diagram 生成一个相应的特征（feature），将其颜色设置为蓝色，分别用深蓝和浅蓝表示。

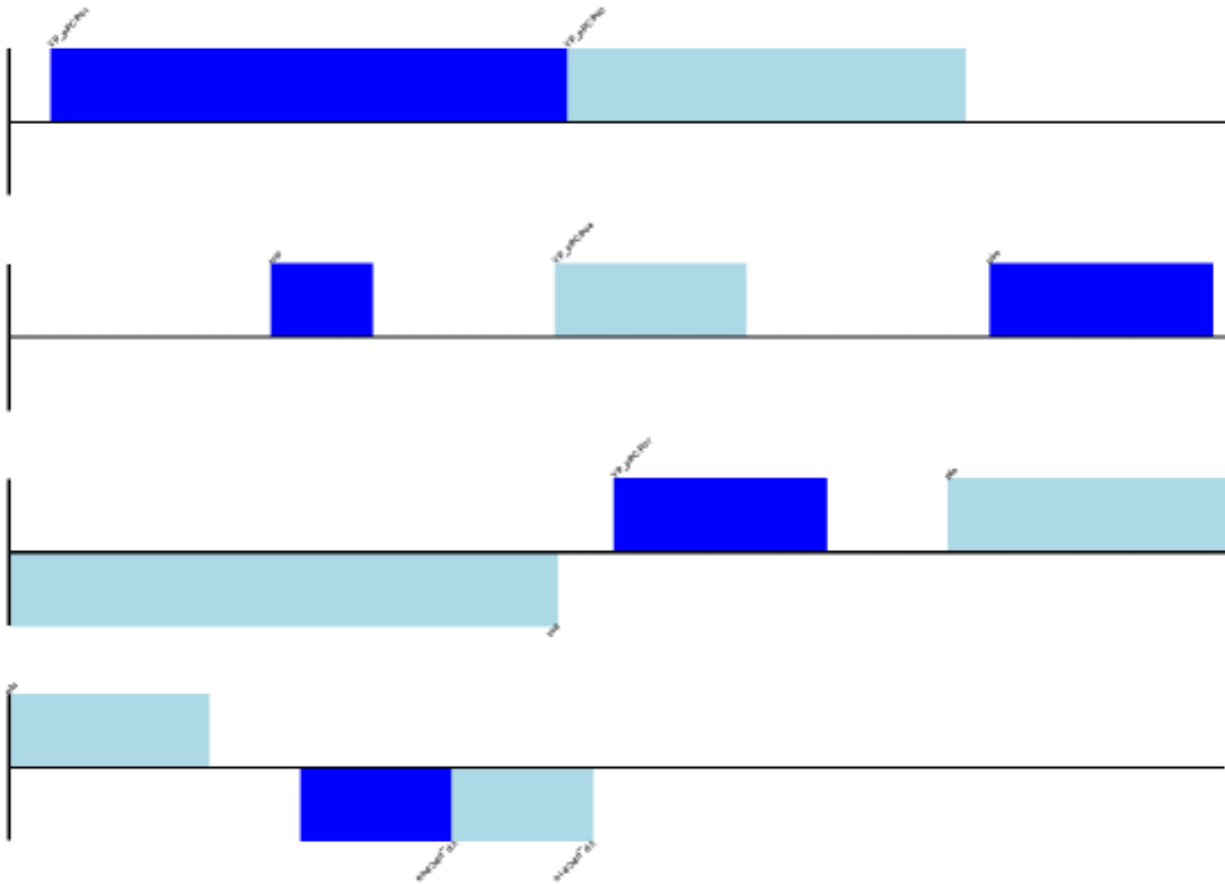
```
for feature in record.features:
    if feature.type != "gene":
        #Exclude this feature
        continue
    if len(gd_feature_set) % 2 == 0:
        color = colors.blue
    else:
        color = colors.lightblue
    gd_feature_set.add_feature(feature, color=color, label=True)
```

创建导出文件需要两步，首先是 draw 方法，它用 ReportLab 对象生成全部图形。然后是 write 方法，将图形存储到格式文件。注意：输出文件格式不止一种。

```
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")
```

如果安装了依赖关系（Dependencies），也可以生成位图（Bitmap image），代码如下：

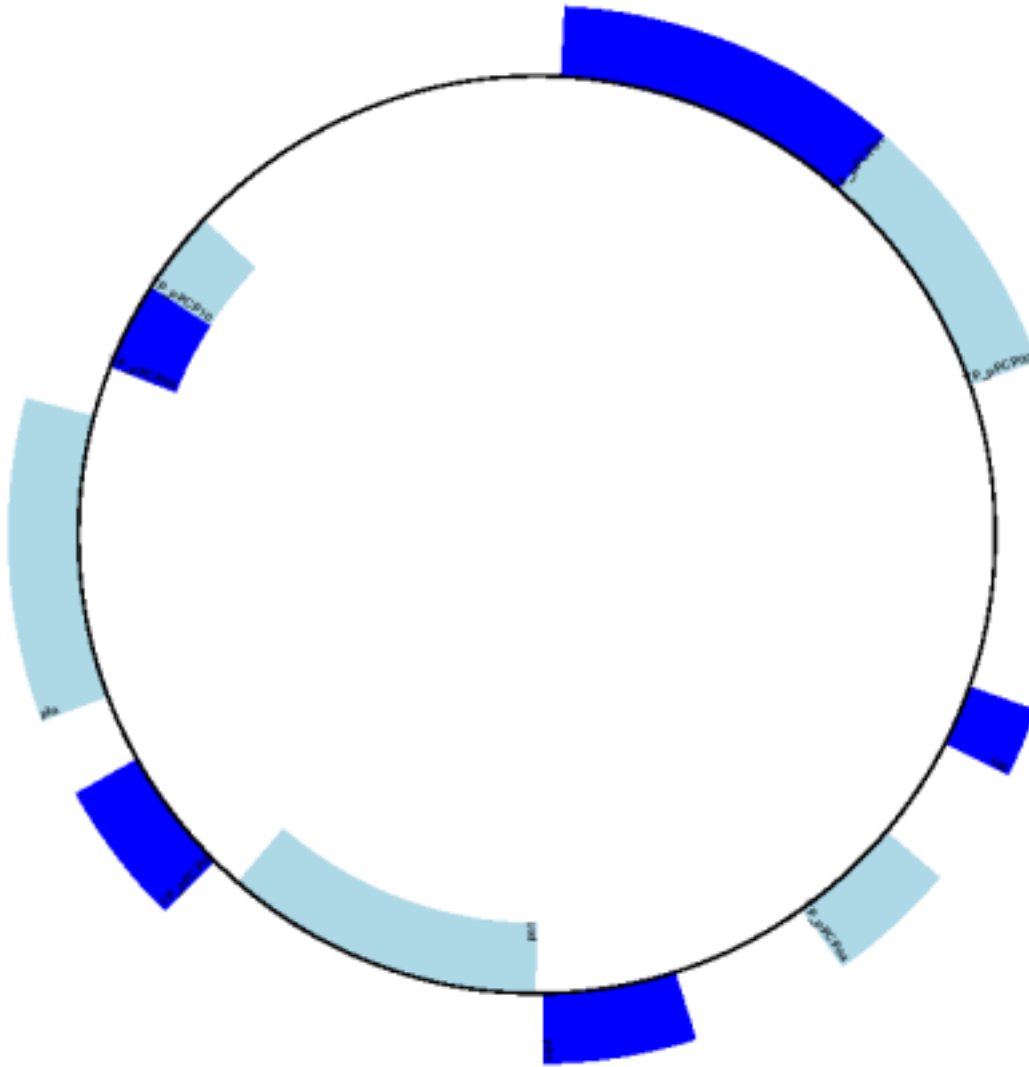
```
gd_diagram.write("plasmid_linear.png", "PNG")
```



注意，我们将代码中的 `fragments` 变量设置为“4”，基因组就会被分为“4”个片段。

如果想要环形图，可以试试以下的代码：

```
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")
```

示例图不是非常精彩，但这仅仅是精彩的开始。

17.1.4 自下而上的实例

现在，用“自下而上”的方法来创建相同的图形。首先新建不同的对象（可以是任何顺序），然后将其组合。

```
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")

#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)

```

同样，利用 `draw` 和 `write` 方法来创建线形图或者环形图，结果应该完全相同（“draw”和“write”部分的代码见 17.1.3）。

17.1.5 简单的 Feature

以上示例中，创建 `diagram` 使用的 `SeqRecord` 的 `SeqFeature` 对象（详见 4.3 章节）。如果你不需要 `SeqFeature` 对象，只将目标 `feature` 定位在坐标轴，仅需要创建 `minimal SeqFeature` 对象，方法很简单，代码如下：

```

from Bio.SeqFeature import SeqFeature, FeatureLocation
my_seq_feature = SeqFeature(FeatureLocation(50,100),strand=+1)

```

对于序列来说，+1 代表正向，-1 代表反向，None 代表两者都有，下面举个简单的示例：

```

from Bio.SeqFeature import SeqFeature, FeatureLocation
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(FeatureLocation(25, 125), strand=+1)
gds_features.add_feature(feature, name="Forward", label=True)
feature = SeqFeature(FeatureLocation(150, 250), strand=None)
gds_features.add_feature(feature, name="Strandless", label=True)
feature = SeqFeature(FeatureLocation(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")

```

图形示例结果请见下一节图中的第一个图，缺省的 `feature` 为浅绿色。

注意，这里用 `name` 参数作为 `feature` 的“说明文本”(caption text)。下文将会讲述更多细节。

17.1.6 Feature 说明

下面代码中，`feature` 作为 `SeqFeature` 的对象添加到 `diagram`。

```
gd_feature_set.add_feature(feature, color=color, label=True)
```

前面的示例用 SeqFeature 的注释为 feature 做了恰当的文字说明。SeqFeature 对象的限定符 (qualifiers dictionary) 缺省值是：gene, label, name, locus_tag, 和 product。简单地说，你可以定义一个名称：

```
gd_feature_set.add_feature(feature, color=color, label=True, name="My Gene")
```

每个 feature 标签的说明文本可以设置字体、位置和方向。说明文本默认的位置在图形符号 (sigil) 的左边，可选择在中间或者右边，线形图中文本的默认方向是 45 旋转。

```
#Large font, parallel with the track
```

```
gd_feature_set.add_feature(feature, label=True, color="green",
                           label_size=25, label_angle=0)
```

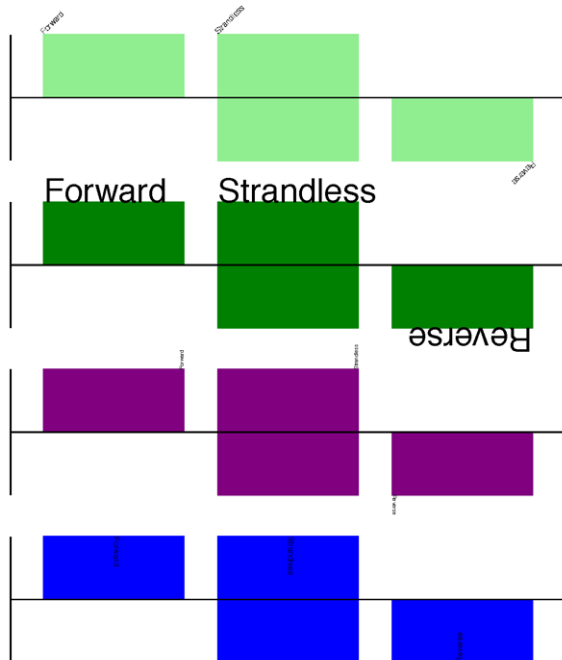
```
#Very small font, perpendicular to the track (towards it)
```

```
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)
```

用前面示例的代码将这三个片段组合之后应该可以得到如下的结果：



除此之外，还可以设置“label_color”来调节标签的颜色（第17.1.9节也将用到这一步），这里没有进行演示。

示例中默认的字体很小，这是比较明智的，因为通常我们会把许多 Feature 同时展示，而不像这里只展示了几个比较大的 feature。

17.1.7 表示 Feature 的图形符号

以上示例中 Feature 的图形符号 (sigil) 默认是一个方框 (plain box), GenomeDiagram 第一版中只有这一选项, 后来 GenomeDiagram 被整合到 Biopython1.50 时, 新增了箭头状的图形符号 (sigil)。

```
#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")
```

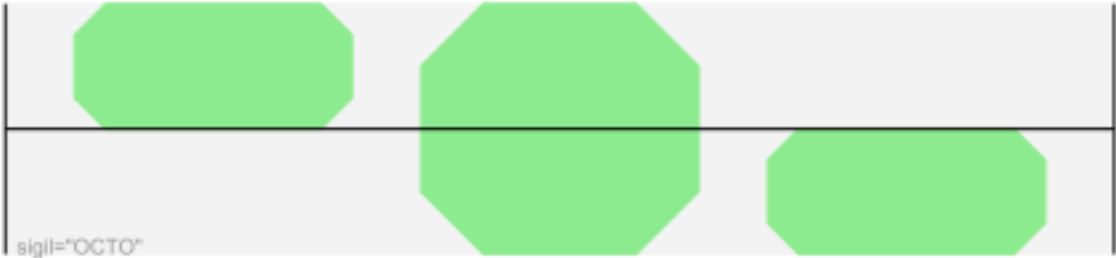
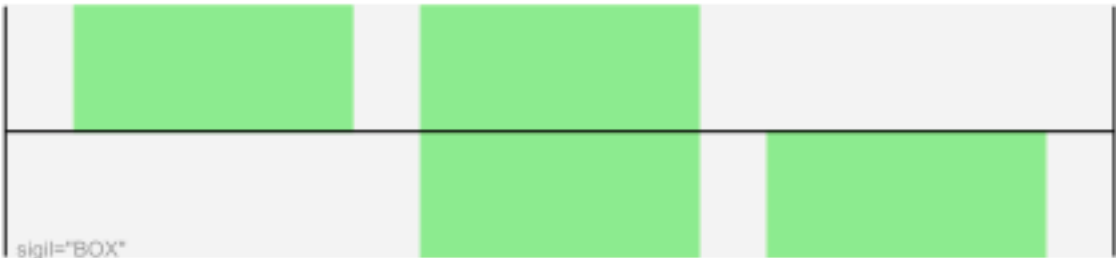
Biopython 1.61 又新增 3 个图形形状 (sigil)。

```
#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)
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")
```

下面就是这些新增的图形形状 (sigil), 多数的图形形状都在边界框 (bounding box) 内部, 在坐标轴的上/下位置代表序列 (Strand) 方向的正/反向, 或者上下跨越坐标轴, 高度是其他图形形状的两倍。“BIGARROW”有所不同, 它总是跨越坐标轴, 方向由 feature 的序列决定。



17.1.8 箭头形状

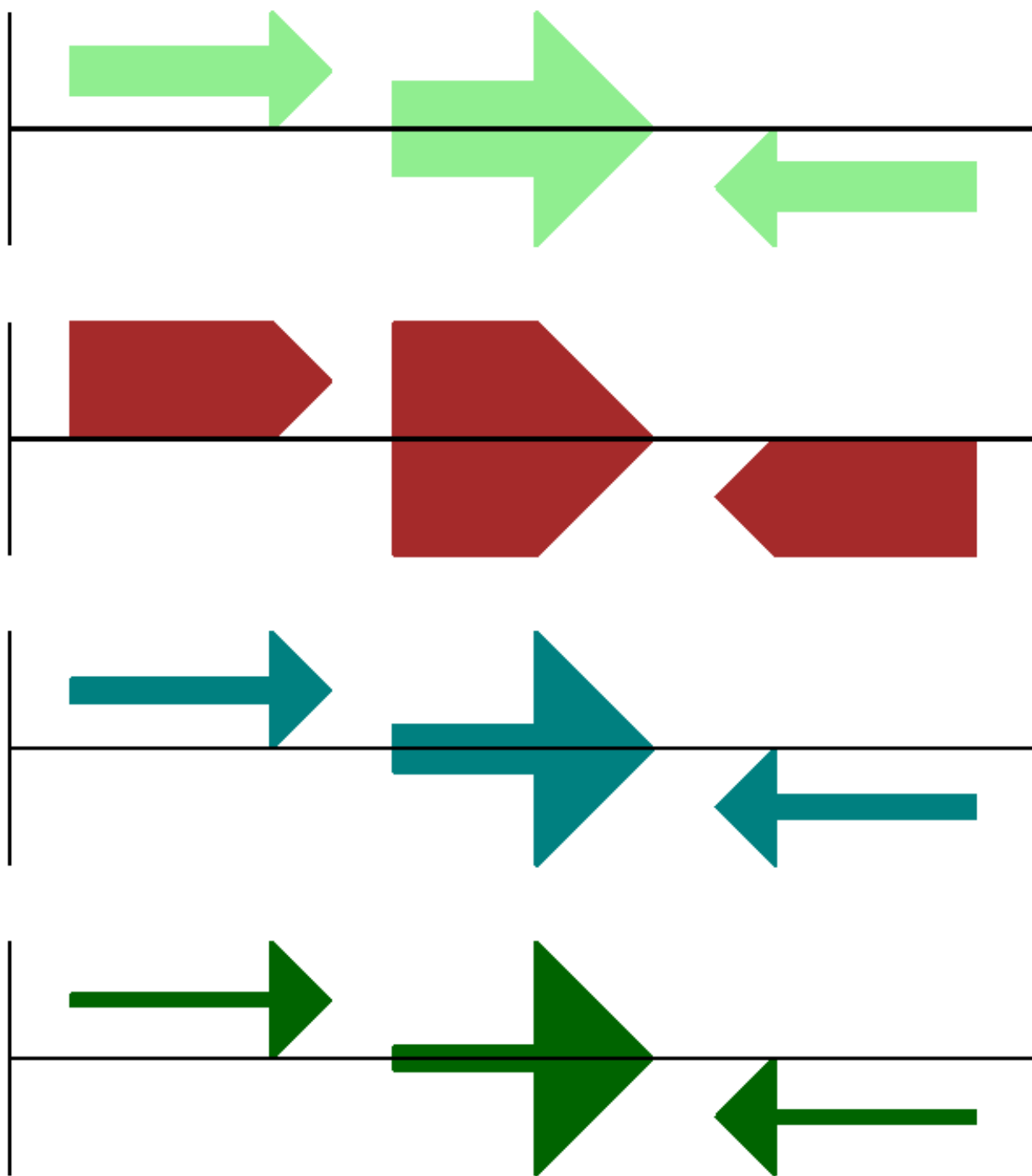
上一部分我们简单引出了箭头形状。还有两个选项可以对箭头形状进行设置：首先根据边界框的高度比例来设置箭杆宽度。

```
#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)
```

结果见下图：



其次，根据边界框的高度比例设置箭头长度（默认为 0.5 或 50%）：

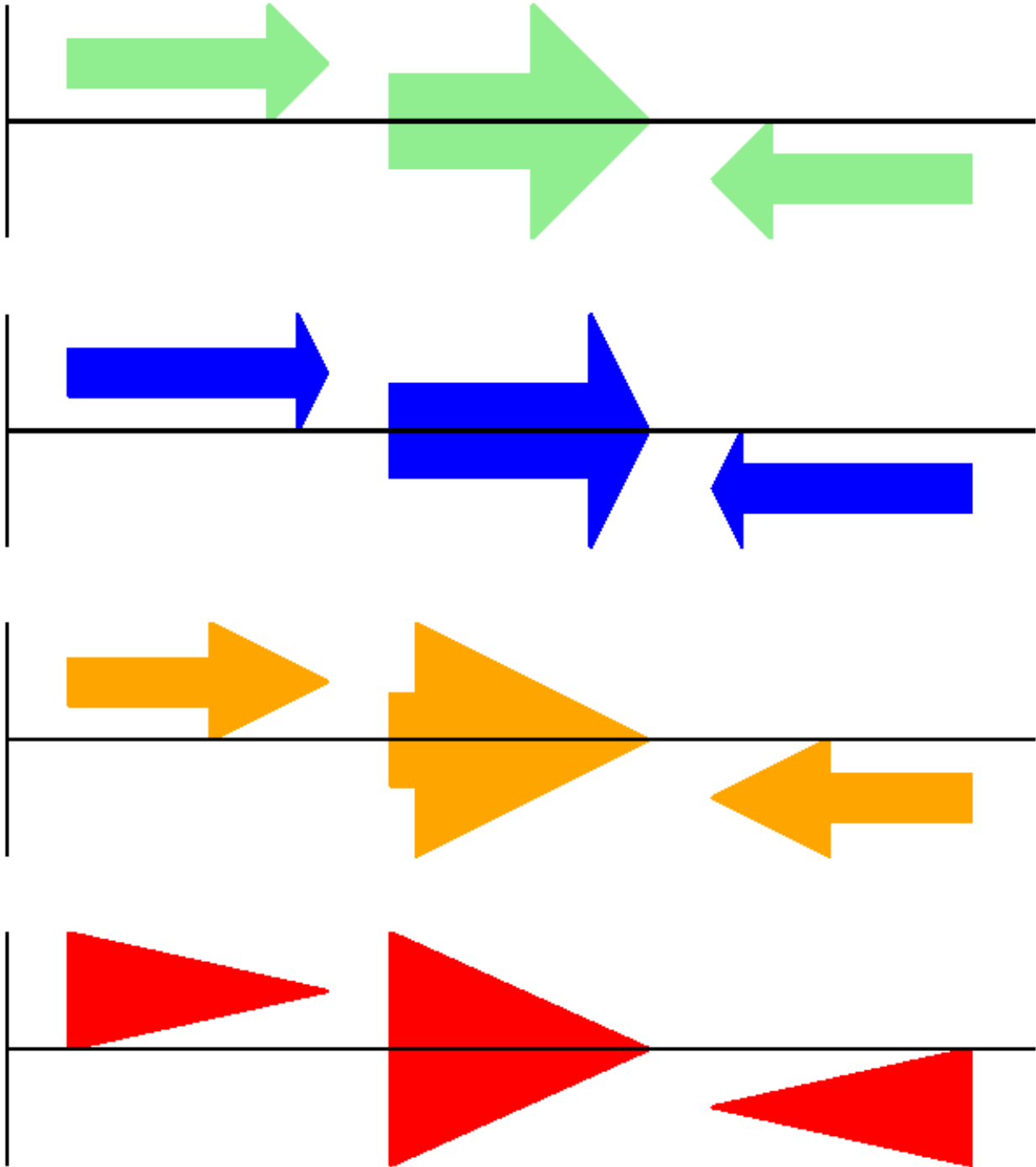
```
#Short arrow heads:
gd_feature_set.add_feature(feature, sigil="ARROW", color="blue",
                           arrowhead_length=0.25)

#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)
```

结果见下图：



Biopython1.61 新增 BIGARROW 箭头形状，它经常跨越坐标轴，箭头指向“左边”代表“反向”，指向“右边”代表“正向”。

```
#A large arrow straddling the axis:
gd_feature_set.add_feature(feature, sigil="BIGARROW")
```

上述 ARROW 形状中的箭杆和箭头设置选项都适用于 BIGARROW。

17.1.9 完美示例

回到“自上而下的示例 Section 17.1.3 中鼠疫杆菌 *Yersinia pestis* biovar *Microtus* 的 pPCP1 质粒，现在使用“图形符号”的高级选项。箭头表示基因，窄框穿越箭头表示限制性内切酶的切割位点。

```
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, FeatureLocation

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(FeatureLocation(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")

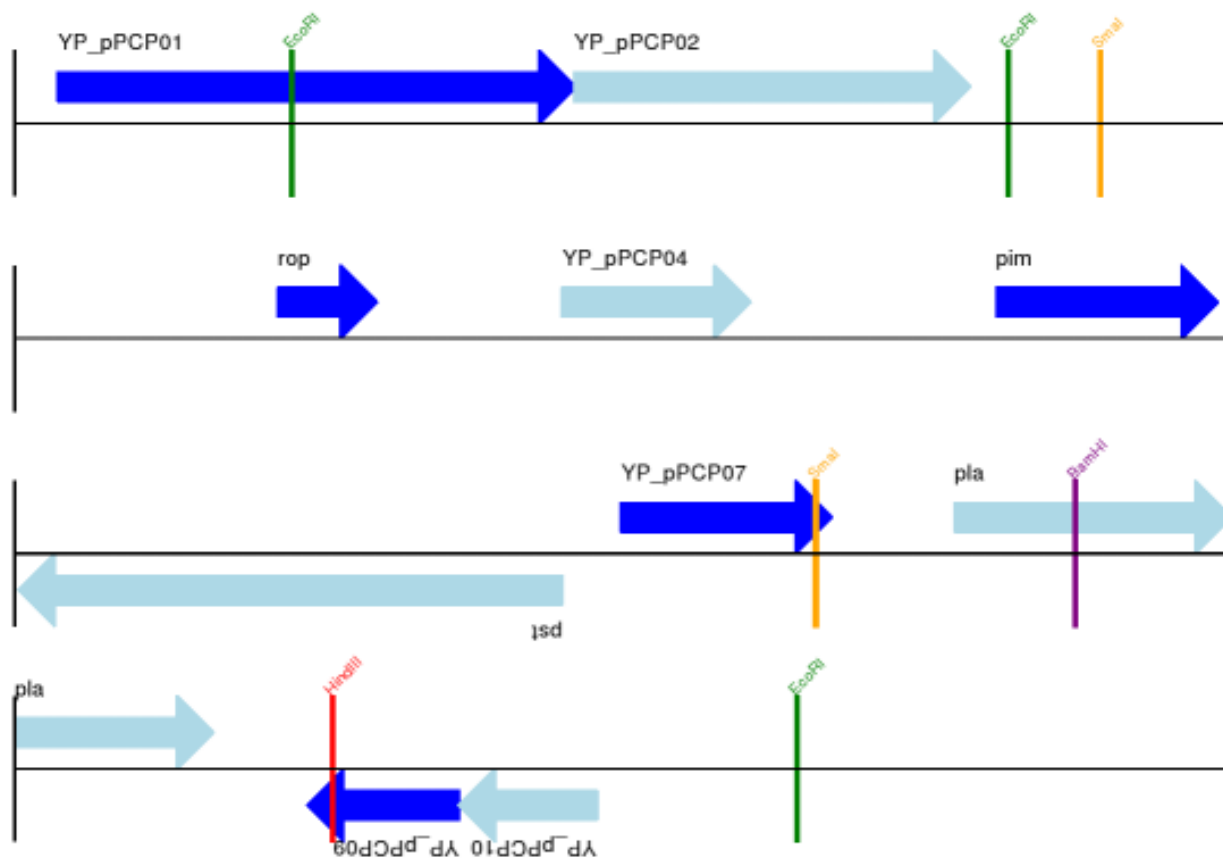
gd_diagram.draw(format="circular", circular=True, pagesize=(20*cm,20*cm),
```

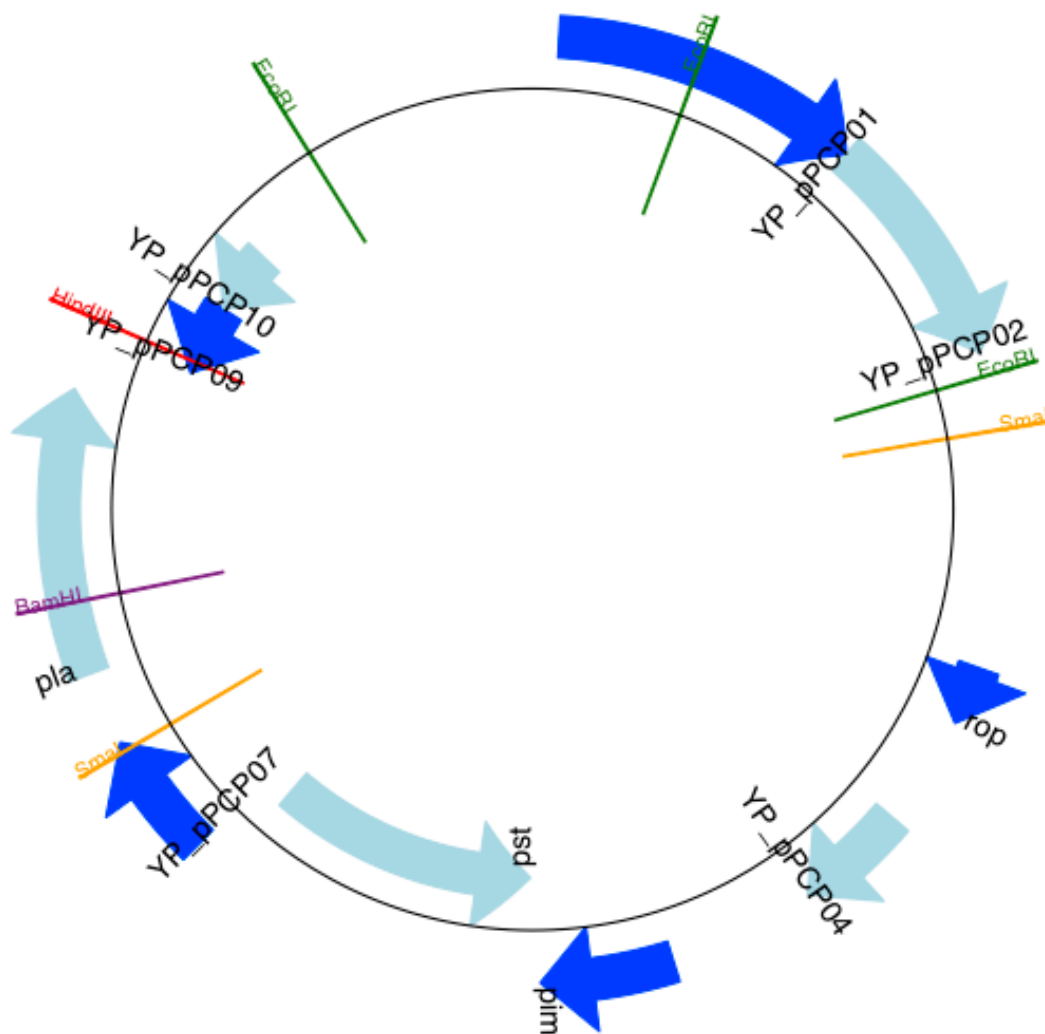
```

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")

```

输出结果见下图：





17.1.10 多重轨迹

前面实例中都是单独的 track，我们可以创建多个 track，比如，一个 track 展示基因，另一个 track 展示重复序列。Proux 等人 2002 年报道的文章 [5] 中图 6 是一个很好的范例，下面我们将三个噬菌体基因组依次进行展示。首先需要三个噬菌体的 GenBank 文件。

- NC_002703 – *Lactococcus* phage Tuc2009, 全基因组大小 (38347 bp)
- AF323668 – Bacteriophage bIL285, 全基因组大小 (35538 bp)
- NC_003212 – *Listeria innocua* Clip11262, 我们将仅关注前噬菌体 5 的全基因组 (长度大体相同)。

这三个文件可以从 Entrez 下载，详情请查阅 9.6。从三个噬菌体基因组文件中分离 (slice) 提取相关 Features 信息 (请查阅 4.6)，保证前两个噬菌体的反向互补链与其起始点对齐，再次保存 Feature (详情请查阅 4.8)。

```
from Bio import SeqIO
```

```
A_rec = SeqIO.read("NC_002703.gb", "gb")
```

```
B_rec = SeqIO.read("AF323668.gb", "gb")
```

```
C_rec = SeqIO.read("NC_003212.gb", "gb")[2587879:2625807].reverse_complement(name=True)
```

图像中用不同颜色表示基因功能的差异。这需要编辑 GenBank 文件中每一个 feature 的颜色参数——就像用 *Sanger's Artemis editor* 处理——才能被 *GenomeDiagram* 识别。但是，这里只需要硬编码 (hard code) 三个颜色列表。

上述 GenBank 文件中的注释信息与 Proux 所用的文件信息并不完全相同，他们还添加了一些未注释的基因。

```
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
```

接下来是“draw”方法，给 diagram 添加 3 个 track。我们在示例中设置不同的开始/结束值来体现它们之间长度不等 (*Biopython* 1.59 及更高级的版本)。

```
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
```

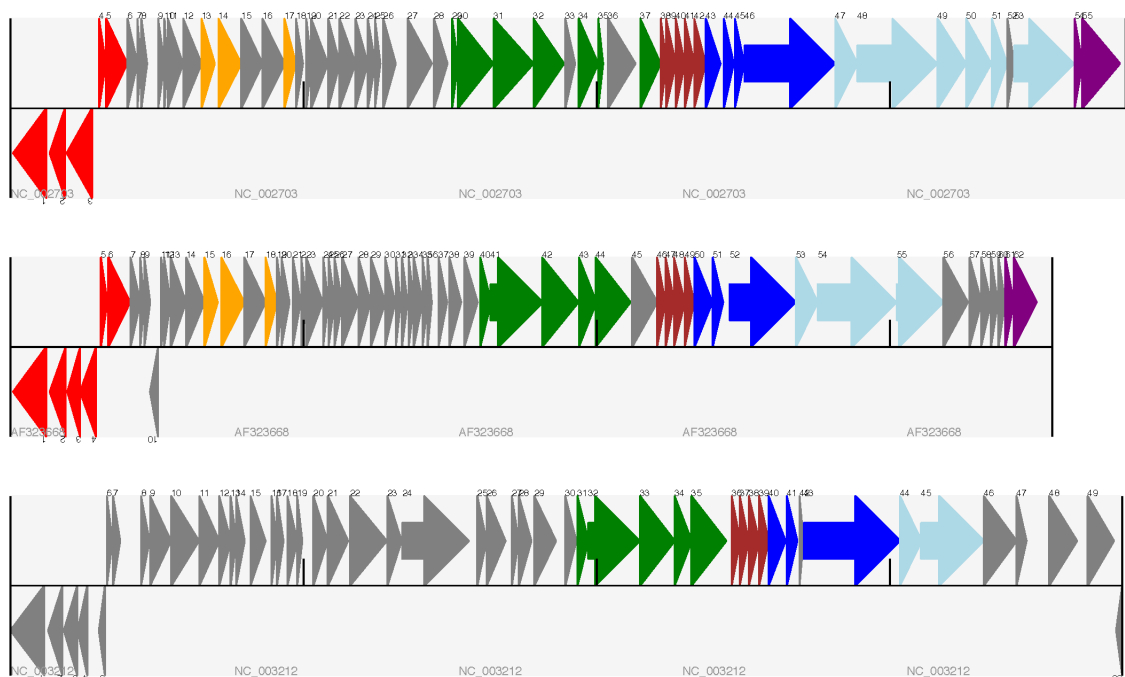
```
gd_diagram.draw(format="linear", pagesize='A4', fragments=1,
    start=0, end=max_len)
```

```
gd_diagram.write(name + ".pdf", "PDF")
```

```
gd_diagram.write(name + ".eps", "EPS")
```

```
gd_diagram.write(name + ".svg", "SVG")
```

结果如图所示：



在示例图中底部的噬菌体没有红色或橙色的基因标记。另外，三个噬菌体可视化图的长度不同，这是因为它们的比例相同，长度却不同。

另外有一点不同，不同噬菌体的同源蛋白质之间用有颜色的 links 相连，下一部分将解决这个问题。

17.1.11 不同 Track 之间的 Cross-Links

Biopython 1.59 新增绘制不同 track 之间 Cross-Links 的功能，这个功能可用于将要展示的简单线形图中，也可用于将线形图分割为短片段 (fragments) 和环形图。

我们接着模仿 Proux 等人 [5] 的图像，我们需要一个包含基因之间的“cross links”、“得分”或“颜色”的列表。实际应用中，可以从 BLAST 文件自动提取这些信息，这里是手动输入的。

噬菌体的名称同样表示为 A, B 和 C。这里将要展示的是 A 与 B 之间的 links，噬菌体 A 和 B 基因的相似百分比存储在元组中。

```
#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"),
]
```

对噬菌体 B 和 C 做同样的处理：

```
#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"),
]
```

噬菌体 A 和 C 的标识符 (Identifiers) 是基因座标签 (locus tags)，噬菌体 B 没有基因座标签，这里用基因名称来代替。以下的辅助函数可用基因座标签或基因名称来寻找 Feature。

```
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)
```

现在将这些标识符对 (identifier pairs) 的列表转换为“SeqFeature”列表，因此来查找它们的坐标定位。现在将下列代码添加到上段代码中 `gd.diagram.draw(...)` 这一行之前，将 cross-links 添加到图像中。示例中的脚本文件 `Proux.et.al.2002.Figure.6.py` 在 Biopython 源程序文件夹的 `Doc/examples` 目录下。

```
from Bio.Graphics.GenomeDiagram import CrossLink
from reportlab.lib import colors
#Note it might have been clearer to assign the track numbers explicitly...
```

```

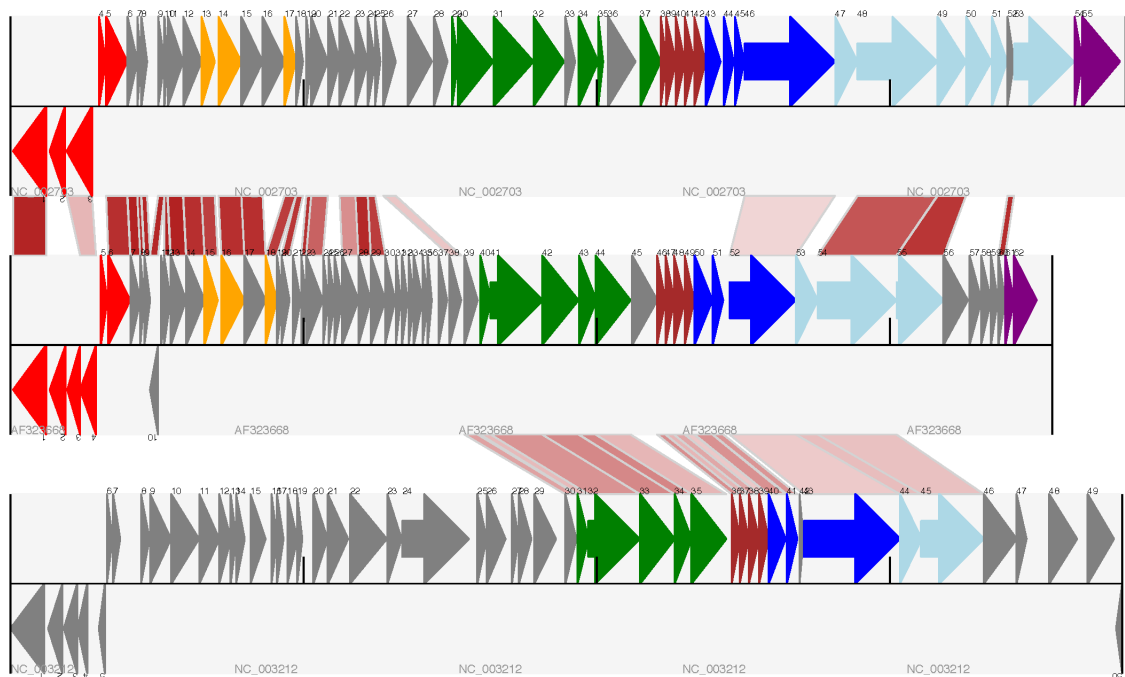
for rec_X, tn_X, rec_Y, tn_Y, X_vs_Y in [(A_rec, 3, B_rec, 2, A_vs_B),
                                         (B_rec, 2, C_rec, 1, B_vs_C)]:

    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)

```

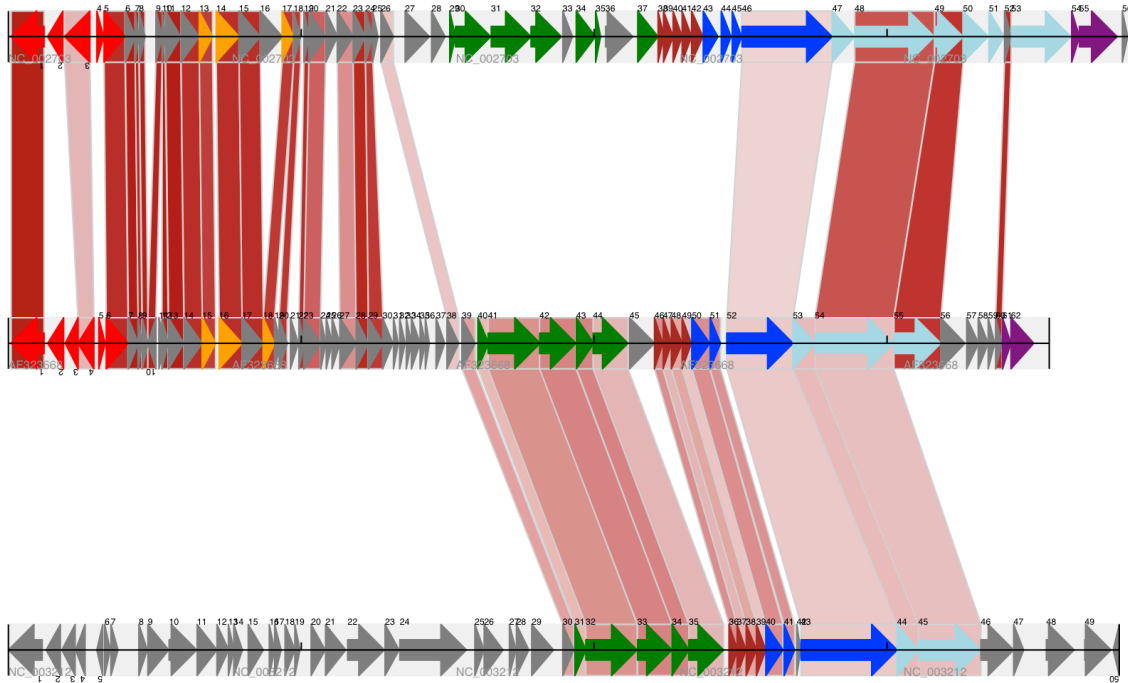
这段代码有几个要点，第一，GenomeDiagram 对象有一个 `cross_track_links` 属性，这个属性只是 `CrossLink` 对象的一组数据。每个 `CrossLink` 对象有两个 track-specific 坐标，示例中用元组 (tuples) 来展现，可用 `GenomeDiagram.Feature` 对象来代替。可选择添加颜色和边框颜色，还可以说明这个 link 是否需要翻转，这个功能易于表现染色体异位。

你也可以看我们是如何将 BLAST 中特征百分比 (Percentage Identity Score) 转换为白-红的渐变色 (白-0%，红-100%)。这个实例中没有 cross-links 的重叠，如果有 links 重叠可以用 ReportLab 库中的透明度 (transparency) 来解决，通过设置颜色的 alpha 通道来使用。然而，若同时使用边框阴影和叠加透明度会增加理解的难度。结果见下图：



当然，Biopython 还有很多增强图像效果的方法。首先，这个示例中的 cross links 是蛋白质之间的，被呈现在一个链的固定区域 (strand specific manor)。可以在 feature track 上用 'BOX' sigil 添加背景区域 (background region) 来扩展 cross link 的效果。同样，可以缩短 feature tracks 之间的垂直高度，使用更多的 links 来代替——一种方法是为空的 track 分配空间。此外，在没有大规模基因重叠的情况下，可以用跨越轴线的 "BIGARROW"，这样就为 track 进一步增加了垂直空间。详情请查看 Biopython 源程序的

Doc/examples 目录下的示例脚本文件：`Proux.et.al.2002_Figure.6.py`。结果见下图：



除此之外，你可能希望在图像编辑软件里手动调整 gene 标签的位置，添加特定标识，比如强调某个特别的区域。

如果有多个叠加的 links，使用 ReportLab 库里的颜色透明度 (transparent color) 是非常好的方法，由于这个示例没有 cross-link 的重叠，所以没有用到颜色透明度 (transparent color)。然而，尽量避免在这个示例中使用边框阴影 (shaded color scheme)。

17.1.12 高级选项

可以通过控制刻度线 (tick marks) 来调节展示比例 (scale)，毕竟每个图形应该包括基本单位和轴线标签的数目。

到目前为止，我们只使用了 `FeatureSet`。`GenomeDiagram` 还可以用 `GraphSet` 来制作线形图，饼状图和 heatmap 热图 (例如在轨迹内展示 feature 中的 GC 含量)。

目前还没有添加这个选项，最后，推荐你去参考 `GenomeDiagram` 单机版 [用户指南 \(PDF\)](#) 和文档字符串 (docstrings)。

17.1.13 转换旧代码

如果你有用 `GenomeDiagram` 独立版本写的旧代码，想将其转换为 Biopython 和新版本可识别的代码，你需要做一些调整——主要是 import 语句。`GenomeDiagram` 的旧版本中使用英式拼写“colour”和“centre”来表示“color”和“center”。被 Biopython 整合后，参数名可以使用任意一种。但是将来可能会不支持英式的参数名。

如果你过去使用下面的方式：

```
from GenomeDiagram import GDFeatureSet, GDDiagram
gdd = GDDiagram("An example")
...
```

你只需要将 import 语句转换成下面这样：

```
from Bio.Graphics.GenomeDiagram import FeatureSet as GDFeatureSet, Diagram as GDDiagram
gdd = GDDiagram("An example")
...
```

希望能够顺利运行。将来你可能想换用新名称，你必须在更大程度上改变你编写代码的方式：

```
from Bio.Graphics.GenomeDiagram import FeatureSet, Diagram
gdd = Diagram("An example")
...
```

or:

```
from Bio.Graphics import GenomeDiagram
gdd = GenomeDiagram.Diagram("An example")
...
```

如果运行过程中出现问题，请到 Biopython 邮件列表中寻求帮助。唯一的缺点就是没有包括旧模块 `GenomeDiagram.GDUtilities`，这个模块有计算 GC 百分比含量的函数，这一部分将会合并到 `Bio.SeqUtils` 模块。

17.2 染色体

`Bio.Graphics.BasicChromosome` 模块可以绘制染色体，Jupe 等人在 2012 发表的文章 [6] 中利用不同的颜色来展示不同的基因家族。

17.2.1 简单染色体

我们用 *Arabidopsis thaliana* 来展示一个简单示例。

首先从 NCBI 的 FTP 服务器 ftp://ftp.ncbi.nlm.nih.gov/genomes/Arabidopsis_thaliana 下载拟南芥已测序的五个染色体文件，利用 `Bio.SeqIO` 函数计算它们的长度。你可以利用 GenBank 文件，但是对于染色体来说，FASTA 文件的处理速度会快点。

```
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)
```

计算出 5 个染色体长度后，就可用 `BasicChromosome` 模块对其作如下的处理：

```
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
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

    #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)

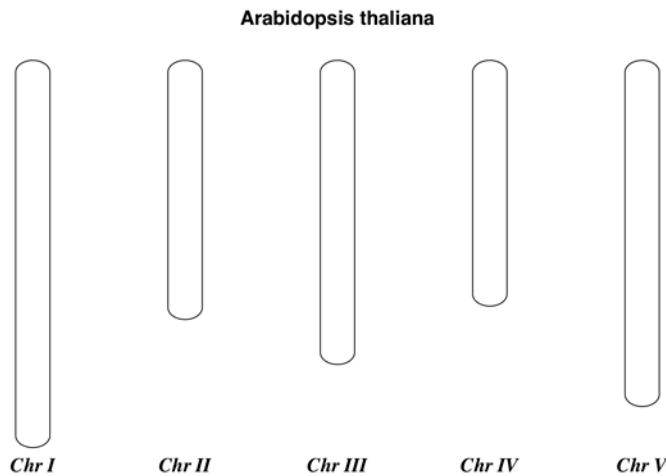
    #Add a closing telomere
    end = BasicChromosome.TelomereSegment(inverted=True)
    end.scale = telomere_length
    cur_chromosome.add(end)

    #This chromosome is done
    chr_diagram.add(cur_chromosome)

chr_diagram.draw("simple_chrom.pdf", "Arabidopsis thaliana")

```

新建的 PDF 文档如图所示：



这个示例可以短小精悍，下面的示例可以展示目标 feature 的定位。

17.2.2 染色体注释

继续前面的示例，我们可以同时展示 tRNA 基因。通过解析 *Arabidopsis thaliana* 的 5 个染色体 GenBank 文件，我们可以对他们进行定位。你需要从 NCBI 的 FTP 服务器下载这些文件 ftp://ftp.ncbi.nlm.nih.gov/genomes/Arabidopsis_thaliana，也可以保存子目录名称或者添加如下的路径：

```
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
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)

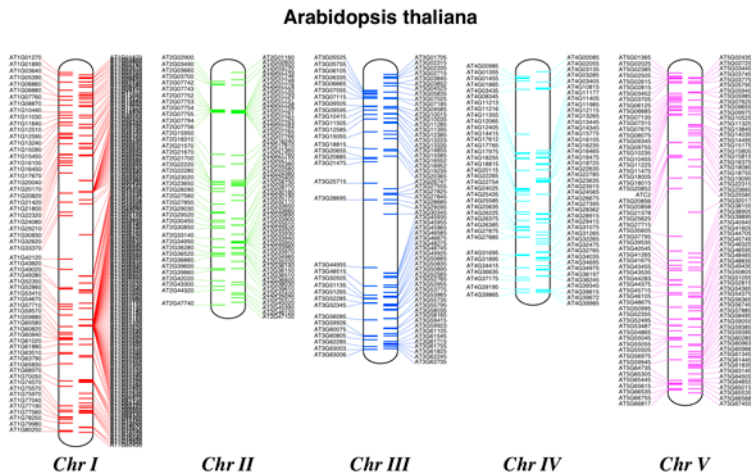
#Add a closing telomere
end = BasicChromosome.TelomereSegment(inverted=True)
end.scale = telomere_length
cur_chromosome.add(end)

#This chromosome is done
chr_diagram.add(cur_chromosome)

chr_diagram.draw("tRNA_chrom.pdf", "Arabidopsis thaliana")

```

如果标签之间太紧密会发出警告，所以要注意第一条染色体的前导链（左手边），可以创建一个彩色的 PDF 文件，如下图所示：



第 18 章 COOKBOOK –用它做一些很酷的事情

Biopython 目前有两个版本的“cookbook”示例——本章（本章包含在教程中许多年，并渐渐成熟），和在 Biopython 维基上的由用户贡献的集合：<http://biopython.org/wiki/Category:Cookbook>。

我们在试着鼓励 Biopython 用户在维基上贡献他们自己的示例。除了能帮助社区之外，分享像这样的示例的一个直接的好处是，你也能从其他 Biopython 用户和开发者中获得一些关于代码的意见反馈——这或许能帮助你改进自己的 Python 代码。

长期来说，我们可能最终会将这一章所有的示例都转移到维基上，或者本教程其他的地方。

18.1 操作序列文件

这部分将展示更多使用第 5 章所描述的 Bio.SeqIO 模块来进行序列输入/输出操作的例子。

18.1.1 过滤文件中的序列

通常你会拥有一个包含许多序列的大文件（例如，FASTA 基因文件，或者 FASTQ 或 SFF 读长文件），和一个包含你所感兴趣的序列的 ID 列表，而你希望创建一个由这一 ID 列表里的序列构成的文件。

让我们假设这个 ID 列表在一个简单的文本文件中，作为每一行的第一个词。这可能是一个表格文件，其第一列是序列 ID。尝试下面的代码：

```
from Bio import SeqIO
input_file = "big_file.sff"
id_file = "short_list.txt"
output_file = "short_list.sff"
wanted = set(line.rstrip("\n").split(None,1)[0] for line in open(id_file))
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)
```

注意，我们使用 Python 的 set 类型而不是 list，这会使得检测成员关系更快。

18.1.2 生成随机基因组

假设你在检视一个基因组序列，寻找一些序列特征——或许是极端局部 GC 含量偏差，或者可能的限制性酶切位点。一旦你使你的 Python 代码在真实的基因组上运行后，尝试在相同的随机化版本基因组上运行，并进行统计分析或许是明智的选择（毕竟，任何你发现的“特性”都可能只是偶然事件）。

在这一讨论中，我们将使用来自 *Yersinia pestis biovar Microtus* 的 pPCP1 质粒的 GenBank 文件。该文件包含在 Biopython 单元测试的 GenBank 文件夹中，或者你可以从我们的网站上得到，[NC_005816.gb](#)。该文件仅有一个记录，所以我们能用 `Bio.SeqIO.read()` 函数把它当做 `SeqRecord` 读入：

```
>>> from Bio import SeqIO
>>> original_rec = SeqIO.read("NC_005816.gb", "genbank")
```

那么，我们怎么生成一个原始序列重排后的版本呢？我会使用 Python 内置的 `random` 模块来做这个，特别是 `random.shuffle` 函数——但是这个只作用于 Python 列表。我们的序列是一个 `Seq` 对象，所以为了重排它，我们需要将它转换为一个列表：

```
>>> import random
>>> nuc_list = list(original_rec.seq)
>>> random.shuffle(nuc_list) #acts in situ!
```

现在，为了使用 `Bio.SeqIO` 输出重排的序列，我们需要使用重排后的列表重新创建包含一个新的 `SeqRecord` 包含随即化后的 `Seq`。要实现这个，我们需要将核苷酸（单字母字符串）列表转换为长字符串——在 Python 中，一般使用字符串对象的 `join` 方法来实现它。

```
>>> from Bio.Seq import Seq
>>> from Bio.SeqRecord import SeqRecord
>>> shuffled_rec = SeqRecord(Seq("".join(nuc_list), original_rec.seq.alphabet),
...                           id="Shuffled", description="Based on %s" % original_rec.id)
```

让我们将所有这些片段放到一起来组成一个完整的 Python 脚本，这个脚本将生成一个 FASTA 序列文件，其包含 30 个原始序列的随机重排版本。

第一个版本只是使用一个大的 `for` 循环，并一个一个的输出记录（使用章节 5.5.4 描述的“`SeqRecord`”的格式化方法）：

```
import random
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

original_rec = SeqIO.read("NC_005816.gb", "genbank")

handle = open("shuffled.fasta", "w")
for i in range(30):
    nuc_list = list(original_rec.seq)
    random.shuffle(nuc_list)
    shuffled_rec = SeqRecord(Seq("".join(nuc_list), original_rec.seq.alphabet), \
                             id="Shuffled%i" % (i+1), \
                             description="Based on %s" % original_rec.id)
    handle.write(shuffled_rec.format("fasta"))
handle.close()
```

我个人更喜欢下面的版本（不使用 `for` 循环），而使用一个函数来重排记录以及一个生成表达式：

```
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), record.seq.alphabet), \
                     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))
handle = open("shuffled.fasta", "w")
SeqIO.write(shuffled_recs, handle, "fasta")
handle.close()
```

18.1.3 翻译 CDS 条目为 FASTA 文件

假设你有一个包含某个物种的 CDS 条目作为输入文件，你想生成一个由它们的蛋白序列组成的 FASTA 文件。也就是，从原始文件中取出每一个核苷酸序列，并翻译它。回到章节 3.9 我们了解了怎么使用 Seq 对象的 translate 方法，和可选的 cds 参数来使得不同的起始密码子能正确翻译。

就像章节 5.5.3 中反向互补的例子中展示的那样，我们可以用 Bio.SeqIO 将与翻译步骤结合起来。对于每一个核苷酸 SeqRecord，我们需要创建一个蛋白的 SeqRecord——并对它命名。

你能编写自己的函数来做这个事情，为你的序列选择合适的蛋白标识和恰当的密码表。在本例中，我们仅使用默认的密码表，并给序列 ID 加一个前缀。

```
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")
```

我们能用这个函数将核苷酸记录转换为蛋白记录，作为输出。一个优雅且内存高效的方式是使用生成表达式 (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")
```

以上代码适用于完整编码序列的 FASTA 文件。如果你使用部分编码序列，你可能希望在以上的例子中使用 nuc_record.seq.translate(to_stop=True)，这会告诉 Biopython 不检查起始密码的有效性，等等。

18.1.4 将 FASTA 文件中的序列变为大写

通常你会从合作者那里得到 FASTA 文件的数据，有时候这些序列可能是大小写混合的。在某些情况下，这些可能是有意为之的（例如，小写的作为低质量的区域），但通常大小写并不重要。你可能希望编辑这个文件以使所有的序列都变得一致（如，都为大写），你可以使用 SeqRecord 对象的 upper() 方法轻易的实现（Biopython 1.55 中引入）：

```
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
```

这是怎么工作的呢？第一行只是导入 Bio.SeqIO 模块。第二行是最有趣的——这是一个 Python 生成器表达式，它提供 mixed.fas 里每个记录的大写版本。第三行中，我们把这个生成器表达式传给 Bio.SeqIO.write() 函数，它会把大写的序列写出到 upper.fas 输出文件。

我们使用生成器（而不是一个列表或列表解析式）的原因是，前一方式每次仅有一个记录保存在内存中。当你在处理包含成千上万的条目的文件时，这可能非常重要。

18.1.5 对序列文件排序

假设你想对一个序列文件按序列长度排序（例如，一个序列拼接的重叠群（contig）集合），而你工作的文件格式可能是像 FASTA 或 FASTQ 这样 Bio.SeqIO 能读写（和索引）的格式。

如果文件足够小，你能将它都一次读入内存为一个 SeqRecord 对象列表，对列表进行排序，并保存它：

```
from Bio import SeqIO
records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(cmp=lambda x,y: cmp(len(x),len(y)))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

唯一巧妙的地方是指明一个比较函数来说明怎样对记录进行排序（这里我们按长度对他们排序）。如果你希望最长的记录在第一个，你可以交换比对，或者使用 reverse 参数：

```
from Bio import SeqIO
records = list(SeqIO.parse("ls_orchid.fasta", "fasta"))
records.sort(cmp=lambda x,y: cmp(len(y),len(x)))
SeqIO.write(records, "sorted_orchids.fasta", "fasta")
```

现在这一实现是非常直接的——但是如果你的文件非常大，你不能像这样把它整个加载到内存中应该怎么办呢？例如，你可能有一些二代测序的读长要根据长度排序。这时你可以使用 Bio.SeqIO.index() 函数解决。

```
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")
```

首先我们使用 Bio.SeqIO.parse() 来将整个文件扫描一遍，并将所有记录的标识和它们的长度保存在一个元组（tuple）中。接着我们对这个元组按照序列长度进行排序，并舍弃这些长度。有了这一排列后的标识列表，Bio.SeqIO.index() 允许我们一个一个获取这些记录，我们把它们传给 Bio.SeqIO.write() 输出。

这些例子都使用 Bio.SeqIO 来解析记录为 SeqRecord 对象，并通过 Bio.SeqIO.write() 输出。当你想排序的文件格式 Bio.SeqIO.write() 不支持应该怎么办呢？如纯文本的 SwissProt 格式。这里有一个额外的解决方法——使用在 Biopython 1.54 (见5.4.2.2) 中 Bio.SeqIO.index() 添加的 get_raw() 方法。

```
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")
handle = open("sorted.fasta", "w")
for id in ids:
    handle.write(record_index.get_raw(id))
handle.close()
```

作为一个奖励，由于以上例子不重复将数据解析为 SeqRecord 对象，所以它会更快。

18.1.6 FASTQ 文件的简单质量过滤

FASTQ 文件格式在 Sanger 被引入，目前被广泛用来存储核苷酸序列 (reads) 和它们的测序质量。FASTQ 文件 (和相关的 QUAL 文件) 是单字母注释 (per-letter-annotation) 的最好的例子，因为序列中每一个核苷酸都有一个相对应的质量分数。任何单字母注释都以 list、tuple 或 string 被保存在 SeqRecord 的 letter_annotations 字典中 (单字母注释具有和序列长度相同个数的元素)。

一个常见的工作是输入一个大的测序读长集合，并根据它们的质量分数过滤它们 (或修剪它们)。下面的例子非常简单，然而足以展示处理 SeqRecord 对象中质量数据的基本用法。我们所有要做的事情是读入一个 FASTQ 文件，过滤并取出那些 PHRED 质量分数在某个阈值 (这里为 20) 以上的序列。

在这个例子中，我们将使用从 ENA 序列读长存档下载的真实数据，<ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz> (2MB)，解压后为 19MB 的文件 SRR020192.fastq。这是在 Roche 454 GS FLX 测序平台生成的感染加利福尼亚海狮的病毒单端数据 (参见 <http://www.ebi.ac.uk/ena/data/view/SRS004476>)。

首先，让我们来统计 reads 的数目：

```
from Bio import SeqIO
count = 0
for rec in SeqIO.parse("SRR020192.fastq", "fastq"):
    count += 1
print "%i reads" % count
```

现在让我们做一个简单的过滤，PHRED 质量不小于 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
```

这只取出了 41892 条读长中的 14580 条。一个更有意义的做法是根据质量来裁剪 reads，但是这里只是作为一个例子。

FASTQ 文件可以包含上百万的记录，所以最好避免一次全部加载它们到内存。这个例子使用一个生成器表达式，这意味着每次只有内存里只有一个 SeqRecord 被创建——避免内存限制。

18.1.7 切除引物序列

在这个例子中，假设我们需要寻找一个 FASTQ 数据中以 GATGACGGTGT 为 5' 端的引物序列的 reads。同上面的例子一样，我们将使用从 ENA 下载的 SRR020192.fastq 文件 (<ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz>)。该方式同样适用于任何其他 Biopython 支持的格式 (例如 FASTA 文件)。

这个代码使用 Bio.SeqIO 和一个生成器表达式 (避免一次加载所有的序列到内存中)，以及 Seq 对象的 startswith 方法来检查读长是否以引物序列开始：

```
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
```

这将从 SRR014849.fastq 找到 13819 条读长记录，并保存为一个新的 FASTQ 文件——with_primer.fastq。

现在，假设你希望创建一个包含这些读长，但去除了所有引物序列的 FASTQ 文件。只需要很小的修改，我们就能对 SeqRecord 进行切片（参见章节 4.6）以移除前 11 个字母（我们的引物长度）：

```
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
```

这也将从 SRR020192.fastq 取出 13819 条读长，但是移除了前十个字符，并将它们保存为另一个新的 FASTQ 文件，with_primer_trimmed.fastq。

最后，假设你想移除部分 reads 中的引物并创建一个新的 FASTQ 文件，而其他的 reads 保持不变。如果我们仍然希望使用生成器表达式，声明我们自己的修剪（trim）函数可能更加清楚：

```
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
```

以上代码会运行较长的时间，因为这次输出文件包含所有 41892 个 reads。再次，我们将使用生成器表达式来避免内存问题。你也可以使用一个生成器函数来替代生成器表达式。

```
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
```

这种形式非常灵活，如果你想做一些更复杂的事情，譬如只保留部分记录——像下一个例子中展示的那样。

18.1.8 切除接头序列

这实际上是前面例子的一个简单扩展。我们将假设 GATGACGGTGT 是某个 FASTQ 格式数据的一个接头序列，并再次使用来自 NCBI 的 SRR020192.fastq 文件 (<ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz>)。

然而在本例中，我们将在读长的 任何位置 查找序列，不仅仅是最开始：

```
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
```

因为我们在这个例子中使用的是 FASTQ 文件，SeqRecord 对象包括 reads 质量分数的单字母注释 (per-letter-annotation)。我们可以通过对具有一定质量分数的 SeqRecord 对象进行切片，并将返回的结果保存到一个 FASTQ 文件。

和上面的例子（只在每个读长的开始查找引物/接头）相比，你会发现有些 reads 剪切后非常短（例如，如果接头序列在读长的中部发现，而不是开始附近）。所以，让我们再加入一个最低长度要求：

```
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
```


通过改变格式名称，你也可以将这个应用于 FASTA 文件。该代码也可以扩展为模糊匹配，而非绝对匹配（或许用一个两两比对，或者考虑读长的质量分数），但是这会使代码变得更慢。

18.1.9 转换 FASTQ 文件

回到章节 5.5.2，我们展示了怎样使用 Bio.SeqIO 来实现两个文件格式间的转换。这里，我们将更进一步探讨二代 DNA 测序中使用的 FASTQ 文件。更加详细的介绍可以参加 Cock *et al.* (2009) [7]。FASTQ 文件同时存储 DNA 序列（以 Python 字符串的形式）和相应的读长质量。

PHRED 分数（在大多数 FASTQ 文件中使用，也存在于 QUAL、ACE 和 SFF 文件中）已经成为一个用来表示某个给定碱基测序错误概率（这里用 P_{e*} 表示）的实际标准（使用一个以 10 为底的对数转换）：

$$Q_{\text{PHRED}} = -10 \times \log_{10}(P_e) \quad (18.0)$$

这意味着一个错误的读长 ($P_{e*} = 1$) 得到的 PHRED 质量为 0，而一个非常好的 $P_{e*} = 0.00001$ 的读长得到的 PHRED 质量为 50。在实际的测序数据中，质量比这个要高的非常稀少，通过后期处理，如读长映射 (mapping) 和组装，PHRED 质量到达 90 是可能的（确实，MAQ 工具允许 PHRED 分数在 0 到 93 之间）。

FASTQ 格式有潜力成为以单文件纯文本方式存储测序读长的字符和质量分数的实际的标准。唯一的美中不足是，目前至少有三个 FASTQ 格式版本，它们相互并不兼容，且难以区分...

1. 原始的 Sanger FASTQ 格式将 PHRED 质量分数和 33 个 ASCII 字符偏移进行编码。NCBI 目前在它们的 Short Read Archive 中使用这种格式。我们在 Bio.SeqIO 中称之为 fastq（或 fastq-sanger）格式。
2. Solexa（后来由 Illumina 收购）引入了他们自己的版本，使用 Solexa 质量分数和 64 个 ASCII 字符偏移进行编码。我们叫做 fastq-solexa 格式。
3. Illumina 工作流 1.3 进一步推出了 PHRED 质量分数（更为一致的版本）的 FASTQ 文件，但是却以 64 个 ASCII 字符偏移编码。我们叫做 fastq-illumina 格式。

Solexa 质量分数采用一种不同的对数转换：

$$Q_{\text{Solexa}} = -10 \times \log_{10} \left(\frac{P_e}{1 - P_e} \right) \quad (18.1)$$

由于 Solexa/Illumina 目前在他们的 1.3 版本的工作流程中已迁移到使用 PHRED 分数，Solexa 质量分数将逐渐淡出使用。如果你将错误估值取等号 (P_{e*})，这两个等式允许在两个评分系统之间进行转换——Biopython 在 Bio.SeqIO.QualityIO 模块中有函数可以实现。这一模块在使用 Bio.SeqIO 进行从 Solexa/Illumina 老文件格式到标准 Sanger FASTQ 文件格式转换时被调用：

```
from Bio import SeqIO
SeqIO.convert("solexa.fastq", "fastq-solexa", "standard.fastq", "fastq")
```

如果你想转换新的 Illumina 1.3+ FASTQ 文件，改变只会导致 ASCII 码的整体偏移。因为尽管编码不同，所有的质量分数都是 PHRED 分数：

```
from Bio import SeqIO
SeqIO.convert("illumina.fastq", "fastq-illumina", "standard.fastq", "fastq")
```

注意，像这样使用 `Bio.SeqIO.convert()` 会比 `Bio.SeqIO.parse()` 和 `Bio.SeqIO.write()` 组合快得多，因为转换 FASTQ（包括 FASTQ 到 FASTA 的转换）的代码经过优化。

对于质量好的读长，PHRED 和 Solexa 分数几乎相等，这意味着，因为 `fasta-solexa` 和 `fastq-illumina` 都使用 64 个 ASCII 字符偏移，它们的文件几乎相同。这是 Illumina 有意设计的，也意味着使用老版本 `fasta-solexa` 格式文件的应用或许也能接受新版本 `fastq-illumina` 格式文件（在高质量的数据上）。当然，两个版本和原始的，由 Sanger、NCBI 和其他地方使用的 FASTQ 标准有很大不同（格式名为 `fastq` 或 `fastq-sanger`）。

了解更多细节，请参见内置的帮助（或 [在线帮助](#)）：

```
>>> from Bio.SeqIO import QualityIO
>>> help(QualityIO)
...
```

18.1.10 转换 FASTA 和 QUAL 文件为 FASTQ 文件

FASTQ 同时 包含序列和他们的质量信息字符串。FASTA 文件 只 包含序列，而 QUAL 文件 只 包含质量。因此，一个单独的 FASTQ 文件可以转换为 成对的 FASTA 和 QUAL 文件，FASTQ 文件也可以由成对的 FASTA 和 QUAL 文件生成。

从 FASTQ 到 FASTA 很简单：

```
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.fasta", "fasta")
```

从 FASTQ 到 QUAL 也很简单：

```
from Bio import SeqIO
SeqIO.convert("example.fastq", "fastq", "example.qual", "qual")
```

然而，反向则有一点复杂。你可以使用 `Bio.SeqIO.parse()` 迭代一个 单独 文件中的所有记录，但是这里我们有两个输入文件。有几个可能的策略，然而这里假设两个文件是真的完全匹配的，最内存高效的方式是同时循环两个文件。代码有些巧妙，所以在 `Bio.SeqIO.QualityIO` 模块中我们提供一个函数来实现，叫做 `PairedFastaQualIterator`。它接受两个句柄（FASTA 文件和 QUAL 文件）并返回一个 `SeqRecord` 迭代器：

```
from Bio.SeqIO.QualityIO import PairedFastaQualIterator
for record in PairedFastaQualIterator(open("example.fasta"), open("example.qual")):
    print record
```

这个函数将检查 FASTA 和 QUAL 文件是否一致（例如，记录顺序是相同的，并且序列长度一致）。你可以和 `Bio.SeqIO.write()` 函数结合使用，转换一对 FASTA 和 QUAL 文件为单独的 FASTQ 文件：

```
from Bio import SeqIO
from Bio.SeqIO.QualityIO import PairedFastaQualIterator
handle = open("temp.fastq", "w") #w=write
records = PairedFastaQualIterator(open("example.fasta"), open("example.qual"))
count = SeqIO.write(records, handle, "fastq")
handle.close()
print "Converted %i records" % count
```

18.1.11 索引 FASTQ 文件

FASTQ 文件通常非常大，包含上百万的读长。由于数据量的原因，你不能一次将所有的记录加载到内存中。这就是为什么上面的例子（过滤和剪切）以迭代的方式遍历整个文件，每次只查看一个 `SeqRecord`。

然而，有时候你不能使用一个大的循环或迭代器——你或许需要随机获取读长。这里 `Bio.SeqIO.index()` 函数被证明非常有用，它允许你使用名字获取 FASTQ 中的任何读长（参见章节 5.4.2）。

我们将再次使用来自 ENA (<ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR020/SRR020192/SRR020192.fastq.gz>) 的文件 `SRR020192.fastq`，尽管这是一个非常小的 FASTQ 文件，只有不到 50,000 读长：

```
>>> from Bio import SeqIO
>>> fq_dict = SeqIO.index("SRR020192.fastq", "fastq")
>>> len(fq_dict)
41892
>>> fq_dict.keys()[:4]
['SRR020192.38240', 'SRR020192.23181', 'SRR020192.40568', 'SRR020192.23186']
>>> fq_dict["SRR020192.23186"].seq
Seq('GTCCCAGTATTCGGATTGTCTGCCAAAACAATGAAATTGACACAGTTTACAAC...CCG', SingleLetterAlphabet())
```

当在包含 7 百万读长的 FASTQ 文件上测试时，索引大概需要花费 1 分钟，然而获取记录几乎是瞬间完成的。

章节 18.1.5 的例子展示了如何使用 `Bio.SeqIO.index()` 函数来对 FASTA 文件进行排序——这也可以用在 FASTQ 文件上。

18.1.12 转换 SFF 文件

如果你处理 454(Roche) 序列数据，你可能会接触 Standard Flowgram Format (SFF) 原始数据。这包括序列读长 (called bases)、质量分数和原始流信息。

一个最常见的工作是转换 SFF 文件为一对 FASTA 和 QUAL 文件，或者一个单独的 FASTQ 文件。这可以使用 `Bio.SeqIO.convert()` 来轻松实现（参见 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
```

注意这个转换函数返回记录的条数，在这个例子中为 10。这将给你 未裁剪 的读长，其中先导和跟随链中低质量的序列，或接头序列将以小写字母显示。如果你希望得到 裁剪 后的读长（使用 SFF 文件中的剪切信息），可以使用下面的代码：

```
>>> 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
```

如果你使用 Linux，你可以向 Roche 请求一份“脱离仪器 (off instrument)”的工具（通常叫做 Newbler 工具）。它提供了另一种的方式来在命令行实现 SFF 到 FASTA 或 QUAL 的转换（但并不支持 FASTQ 输出）。


```
$ 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
```

Biopython 以大小写混合的方式来表示剪切位点，这是有意模拟 Roche 工具的做法。

要获得 Biopython 对 SFF 支持的更多信息，请参考内部帮助：

```
>>> from Bio.SeqIO import SffIO
>>> help(SffIO)
...
```

18.1.13 识别开放读码框

在识别可能的基因中一个非常简单的第一步是寻找开放读码框 (Open Reading Frame, ORF)。这里我们的意思是寻找六个编码框中所有的没有终止密码子的长区域——一个 ORF 是一个不包含任何框内终止密码子的核苷酸区域。

当然，为了发现基因，你也需要确定起始密码子、可能的启动子的位置——而且在真核生物中，你也需要关心内含子。然而，这种方法在病毒和原核生物中仍然有效。

为了展示怎样用 Biopython 实现这个目的，我们首先需要有一个序列来查找。作为例子，我们再次使用细菌的质粒——尽管这次我们将以没有任何基因标记的纯文本 FASTA 文件开始：[NC_005816.fna](#)。这是一个细菌序列，所以我们需要使用 NCBI 密码子表 11（参见章节 3.9 关于翻译的介绍）。

```
>>> from Bio import SeqIO
>>> record = SeqIO.read("NC_005816.fna", "fasta")
>>> table = 11
>>> min_pro_len = 100
```

这里有一个巧妙的技巧，使用 Seq 对象的 split 方法获得一个包含六个读码框中所有可能的 ORF 翻译的列表：

```
>>> 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
KSGELRQTPPASSTLHLRLILQRSGVMEL...NPE - length 285, strand 1, frame 1
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, frame 1
VKKILYIKALFLCTVIKLRFFSVNMMKF...DLP - length 165, strand 1, frame 1
NQIQGVICSPDSGEFMTVFETVMEIKILHK...GVA - length 355, strand 1, frame 2
RRKEHVSCKRRPQKRPRRRFFHRLRPPDE...PTR - length 128, strand 1, frame 2
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, frame 2
QGGSYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, frame 0
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, frame 0
WGKLQVIGLSMWMVLSQRFDWLNEQEDA...ESK - length 125, strand -1, frame 1
RGIFMSDTMNVNGSGGVPAFLFSGSTLSSY...LLK - length 361, strand -1, frame 1
WDVKTVTGVLHHPFHLTFSLCPGATQSGR...VKR - length 111, strand -1, frame 1
LSHTVTDTFDQMAQVGLCQCVNVLDEVTG...KAA - length 107, strand -1, frame 2
RALTGLSAPGIRSQTSCDRRLRELYVPVSL...PLQ - length 119, strand -1, frame 2
```

注意，这里我们从 每条 序列的 5'末（起始）端开始计算读码框。对 正向 链一直从 5'末（起始）端开始计算有时更加容易。

你可以轻易编辑上面的循环代码，来创建一个待选蛋白列表，或者将它转换为一个列表解析。现在，这个代码所不能做的一个事情是记录蛋白的位置信息。

你可以用几种方式来处理。例如，下面的代码以蛋白计数的方式记录位置信息，并通过乘以三倍来转换为父序列（parent sequence），并记录编码框和链的信息：

```
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 = str(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)
```

输出是：

```
NQIQGVICSPDSGEFMVTFETVMEIKILHK...GVA - length 355, strand 1, 41:1109
WDVKTVTGVLHHPFHLTFSLCPEGATQSGR...VKR - length 111, strand -1, 491:827
KSGELRQTPPASSTLHLRLILQRSGVMME...NPE - length 285, strand 1, 1030:1888
RALTGLSAPGIRSQTSCDRLRELYVPVSL...PLQ - length 119, strand -1, 2830:3190
RRKEHVSKKRRPQKRPRRRFFHRLRPPDE...PTR - length 128, strand 1, 3470:3857
GLNCSFFSICNWKFIDYINRLFQIIYLCKN...YYH - length 176, strand 1, 4249:4780
RGIFMSDTMVVNGSGGVP AFLFSGSTLSSY...LLK - length 361, strand -1, 4814:5900
VKKILYIKALFLCTVIKLRRFISVNNMKF...DLP - length 165, strand 1, 5923:6421
LSHTVTDFTDQMAQVGLCQCQVNVFLDEV...KAA - length 107, strand -1, 5974:6298
GCLMKKSSIVATIITILSGSANAASSQLIP...YRF - length 315, strand 1, 6654:7602
IYSTSEHTGEQVMRTLDEVIASRSPESQTR...FHV - length 111, strand -1, 7788:8124
WGKLQVIGLSMWMVLSQRFDWLNEQEDA...ESK - length 125, strand -1, 8087:8465
TGKQNSCQMSAIWQLRQNTATKTRQNRARI...AIK - length 100, strand 1, 8741:9044
QSGGYAFPHASILSGIAMSHFYFLVLHAVK...CSD - length 114, strand -1, 9264:9609
```

如果你注释掉排序语句，那么蛋白序列将和之前显示的顺序一样，所以你能确定这是在做相同的事情。

这里，我们可以按位置进行排序，使得和 GenBank 文件中的实际注释相比对较更加容易（就像章节 17.1.9 中显示的那样）。

然而，如果你想要的只是所有开放读码框的位置，翻译每一个可能的密码子将是很浪费时间的，包括转换和查找反向互补链。那么，你所要做的所有事情是查找可能的终止密码子（和他们反向互补）。使用正则表达式是一个很直接的方式（参见 Python 中的 `re` 模块）。这是描述查找字符串的一个非常强大的模块（然而非常复杂），也被许多编程语言和命令行工具，如 `grep`，所支持。你能找到一本书来描述它的使用！

18.2 序列解析与简单作图

这一部分展示更多使用第 5 章介绍的 `Bio.SeqIO` 模块进行序列解析的例子，以及 Python 类库 `matplotlib` 中 `pylab` 的作图接口（参见 [matplotlib 主页的教程](#)）。注意，跟随这些例子，你需要安装 `matplotlib` - 但是即使没有它，你依然可以尝试数据的解析的内容。

18.2.1 序列长度柱状图

在很多时候，你可能想要将某个数据集中的序列长度分布进行可视化——例如，基因组组装项目中的 `contig` 的大小范围。在这个例子中，我们将再次使用我们的兰花 FASTA 文件 `ls_orchid.fasta`，它只包含 94 条序列。

首先，我们使用 `Bio.SeqIO` 来解析这个 FASTA 文件，并创建一个序列长度的列表。你可以用一个 `for` 循环来实现，然而我觉得列表解析（list comprehension）更简洁：

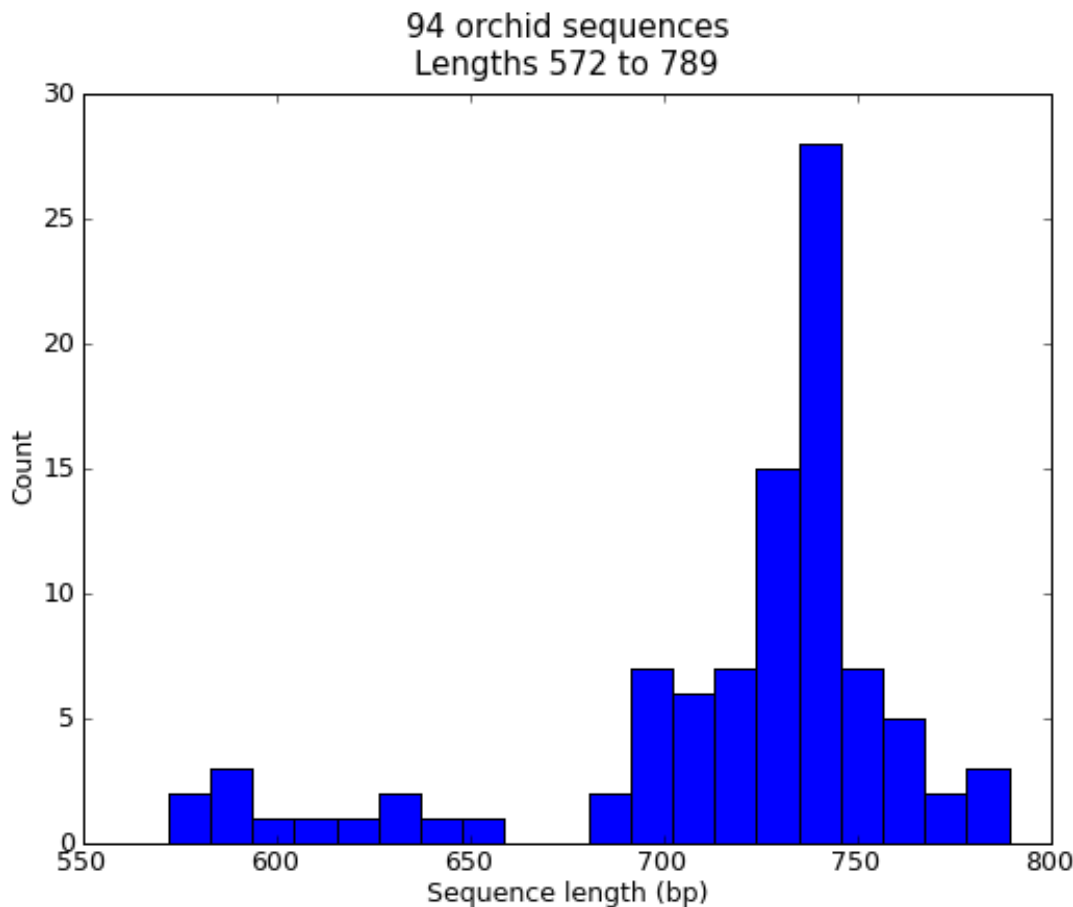
```
>>> from Bio import SeqIO
>>> sizes = [len(rec) for rec in SeqIO.parse("ls_orchid.fasta", "fasta")]
>>> len(sizes), min(sizes), max(sizes)
(94, 572, 789)
>>> sizes
[740, 753, 748, 744, 733, 718, 730, 704, 740, 709, 700, 726, ..., 592]
```

现在我们得到了所有基因的长度（以整数列表的形式），我们可以用 `matplotlib` 的柱状图功能来显示它。

```
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()
```

这将弹出一个包含如下图形的新的窗口：



注意，这些兰花序列的长度大多数大约在 740bp 左右，这里有可能有两个不同长度的序列分类，其中包含一个较短的序列子集。

提示：除了使用 `pylab.show()` 在窗口中显示图像以外，你也可以使用 `pylab.savefig(...)` 来将图像保存为图像文件中（例如 PNG 或 PDF 文件）。

18.2.2 序列 GC% 含量作图

对于核酸序列，另一个经常计算的值是 GC 含量。例如，你可能想要查看一个细菌基因组中所有基因的 GC%，并研究任何离群值来确定可能最近通过基因水平转移而获得的基因。同样，对于这个例子，我们再次使用兰花 FASTA 文件 `ls_orchid.fasta`。

首先，我们使用 `Bio.SeqIO` 解析这个 FASTA 文件并创建一个 GC 百分含量的列表。你可以使用 `for` 循环，但我更喜欢这样：

```
from Bio import SeqIO
from Bio.SeqUtils import GC

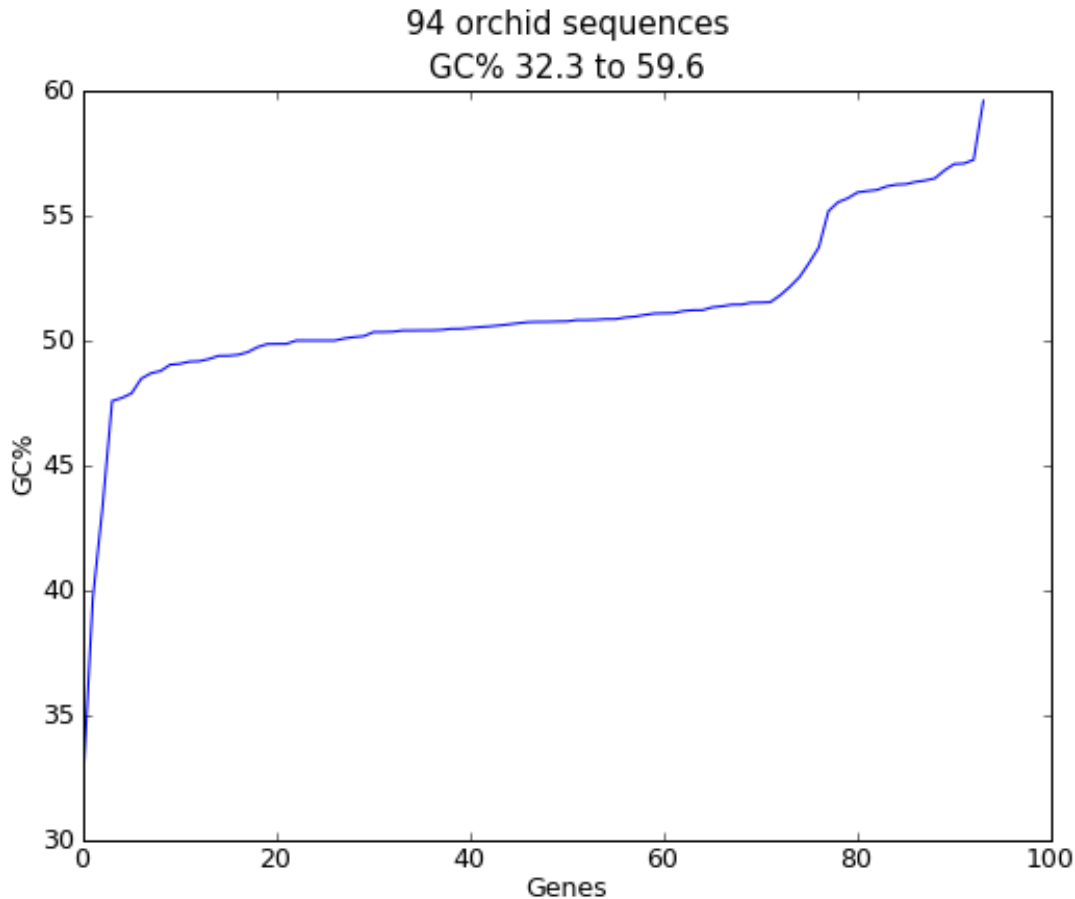
gc_values = sorted(GC(rec.seq) for rec in SeqIO.parse("ls_orchid.fasta", "fasta"))
```

读取完每个序列并计算了 GC 百分比，我们接着将它们按升序排列。现在，我们用 `matplotlib` 来对这个浮点数列表进行可视化：

```
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()
```

像之前的例子一样，弹出一个窗口中将包含如下图形：



如果你使用的是一个物种中的所有基因集，你可能会得到一个更加平滑的图。

18.2.3 核苷酸点线图

点线图是可视化比较两条核苷酸序列的相似性的一种方式。采用一个滑动窗来相互比较较短的子序列（比较通常根据一个不匹配阈值来实现）。为了简单起见，此处我们将只查找完全匹配（如下图黑色所示）。

我们需要两条序列开始。为了论证，我们只取兰花 FASTA 文件中的前两条序列。ls_orchid.fasta:

```
from Bio import SeqIO
handle = open("ls_orchid.fasta")
record_iterator = SeqIO.parse(handle, "fasta")
rec_one = record_iterator.next()
rec_two = record_iterator.next()
handle.close()
```

我们将展示两种方式。首先，一个简单的实现，它将所有滑动窗大小的子序列相互比较，并生产一个相似性矩阵。你可以创建一个矩阵或数组对象，而在这儿，我们只用一个用嵌套的列表解析生成的布尔值列表

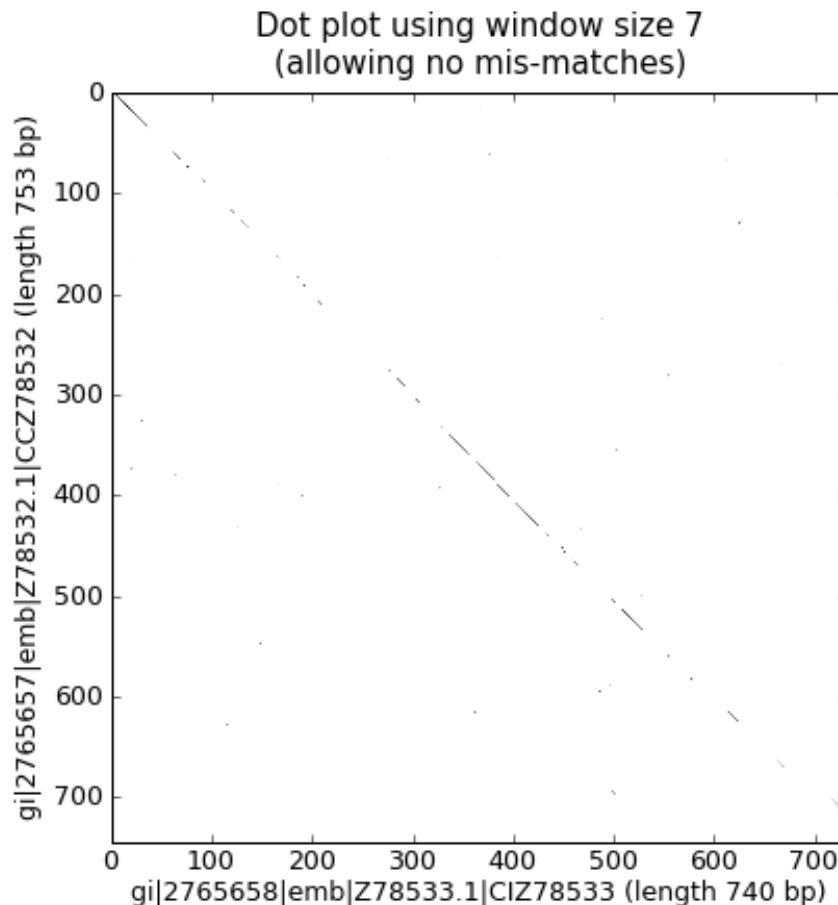
的列表。

```
window = 7
seq_one = str(rec_one.seq).upper()
seq_two = str(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)]
```

注意，我们在这里并没有检查反向的互补匹配。现在我们将使用 matplotlib 的 `pylab.imshow()` 函数来显示这个数据，首先启用灰度模式，以保证这是在黑白颜色下完成的：

```
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()
```

这将弹出一个新的窗口，包含类似这样的图形：



可能如您所料，这两条序列非常相似，图中部分滑动窗大小的线沿着对角线匹配。图中没有对角线外的点，这意味着序列中并没有倒位或其他有趣的偏离对角线匹配。

上面的代码在小的例子中工作得很好，但是应用到大的序列时，这里有两个问题。首先，这种以穷举地进行所有可能的两两比对非常缓慢。作为替代，我们将创建一个词典来映射所有滑动窗大小的子序列

的位置，然后取两者的交集来获得两条序列中都发现的子序列。这将占用更多的内存，然而速度更快。另外，`pylab.imshow()` 函数只能显示较小的矩阵。作为替代，我们将使用 `pylab.scatter()` 函数。

我们从创建，从滑动窗大小的子序列到其位置的字典映射开始：

```
window = 7
dict_one = {}
dict_two = {}
for (seq, section_dict) in [(str(rec_one.seq).upper(), dict_one),
                           (str(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
#(Python 2.3 would require slightly different code here)
matches = set(dict_one).intersection(dict_two)
print "%i unique matches" % len(matches)
```

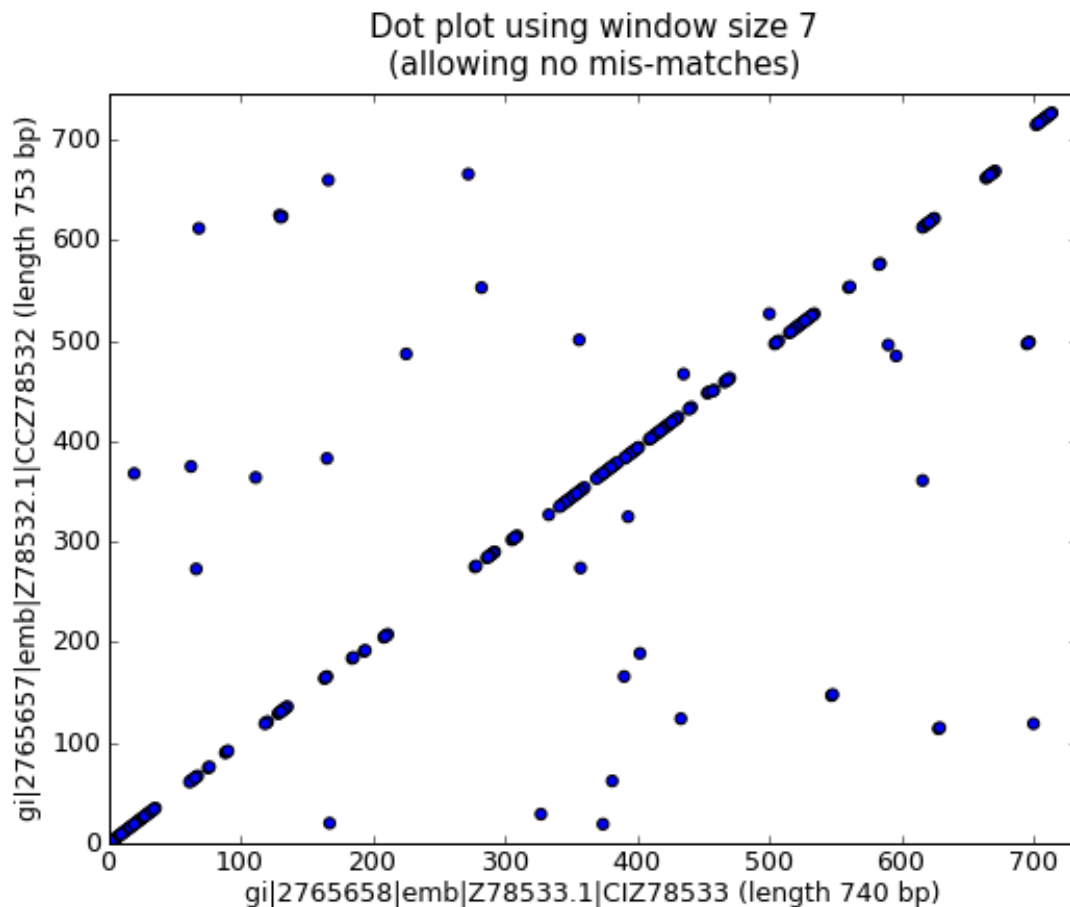
为了使用 `pylab.scatter()` 函数，我们需要两个分别对应 x 和 y 轴的列表：

```
#Create lists of x and y co-ordinates 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)
```

现在我们能以散点图的形式画出优化后的点线图：

```
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()
```

这将弹出一个新的窗口，包含如下图形：



我个人认为第二个图更加易读！再次注意，我们在这里 没有 检查反向互补匹配——你可以扩展这个例子来实现它，或许可以以一种颜色显示正向匹配，另一种显示反向匹配。

18.2.4 绘制序列读长数据的质量图

如果你在处理二代测序数据，你可能希望绘制数据的质量图。这里使用两个包含双末端 (paired end) 读长的 FASTQ 文件作为例子，其中 SRR001666_1.fastq 为正向读长，SRR001666_2.fastq 为反向读长。它们可以从 ENA 序列读长档案的 FTP 站点下载 (ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_1.fastq.gz 和 ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR001/SRR001666/SRR001666_2.fastq.gz)，且来自 *E. coli* ——参见 <http://www.ebi.ac.uk/ena/data/view/SRR001666> 的详细介绍。在下面的代码中，`pylab.subplot(...)` 函数被用来在两个子图中展示正向和反向的质量。这里也有少量的代码来保证仅仅展示前 50 个读长的质量。

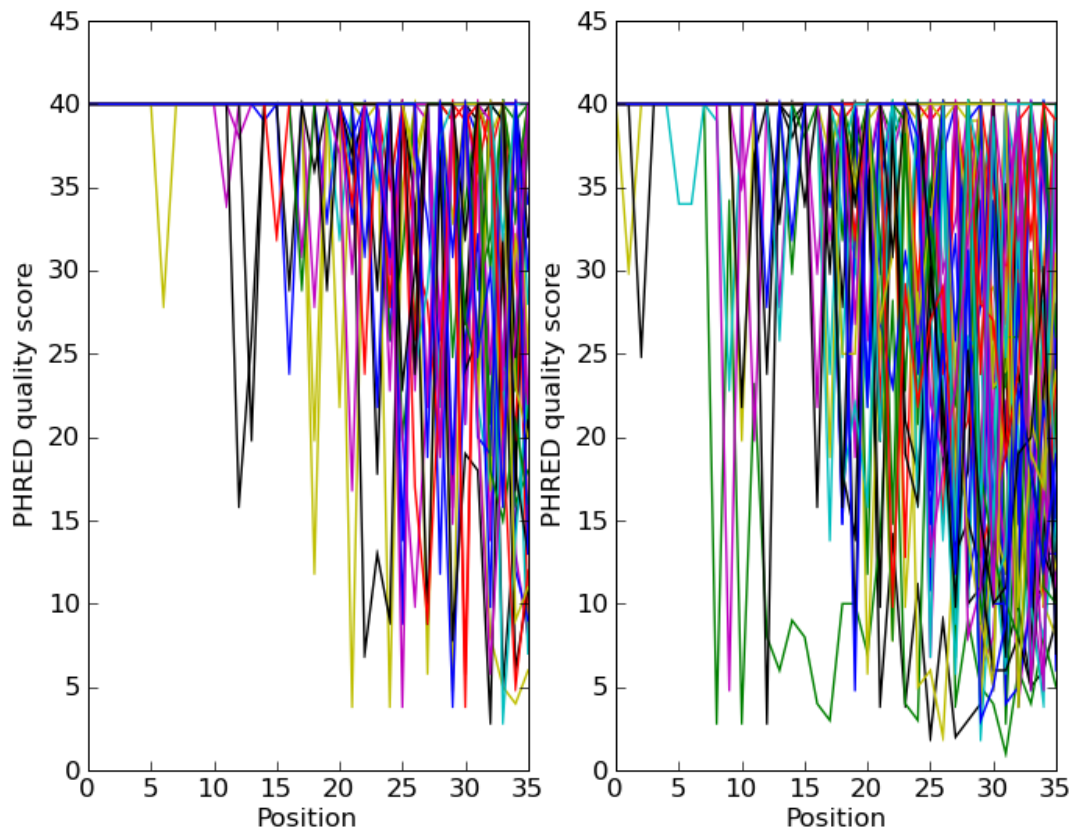
```
import pylab
from Bio import SeqIO
for subfigure in [1,2]:
    filename = "SRR001666_%i.fastq" % subfigure
    pylab.subplot(1, 2, subfigure)
    for i,record in enumerate(SeqIO.parse(filename, "fastq")):
        if i >= 50 : break #trick!
        pylab.plot(record.letter_annotations["phred_quality"])
    pylab.ylim(0,45)
    pylab.ylabel("PHRED quality score")
```



```
pylab.xlabel("Position")
pylab.savefig("SRR001666.png")
print "Done"
```

你应该注意到，这里我们使用了 Bio.SeqIO 的格式名称 fastq，因为 NCBI 使用标准 Sanger FASTQ 和 PHRED 分数的存储这些读长。然而，你可能从读长的长度中猜到，这些数据来自 Illumina Genome Analyzer，而且可能最初是以 Solexa/Illumina FASTQ 两种格式变种中的一种存在。

这个例子使用 pylab.savefig(...) 函数，而不是 “pylab.show(...)”，然而就像前面提到的一样，它们两者都非常有用。下面是得到的结果：



18.3 处理序列比对

这部分可以看做是第6章的继续。

18.3.1 计算摘要信息

一旦你有一个比对，你很可能希望找出关于它的一些信息。我们尽力将这些功能分离到单独的能作用于比对对象的类中，而不是将所有的能生成比对信息的函数都放入比对对象本身。

准备计算比对对象的摘要信息非常快捷。假设我们已经得到了一个比对对象 alignment，例如由在第6章介绍的 Bio.AlignIO.read(...) 读入。我们获得该对象的摘要信息所要做的所有事情是：

```
from Bio.Align import AlignInfo
summary_align = AlignInfo.SummaryInfo(alignment)
```

summary_align 对象非常有用，它将帮你做以下巧妙的事情：

1. 计算一个快速一致序列—参见章节 18.3.2
2. 获取一个针对该比对的位点特异性打分矩阵—参见章节 18.3.3
3. 计算比对的信息量—参见章节 18.3.4
4. 生成该比对中的替换信息—章节 18.4 详细描述了使用该方法生成一个替换矩阵

18.3.2 计算一个快速一致序列

在章节 18.3.1 中描述的 SummaryInfo 对象提供了一个可以快速计算比对的保守 (consensus) 序列的功能。假设我们有一个 SummaryInfo 对象，叫做 summary_align，我们能通过下面的方法计算一个保守序列：

```
consensus = summary_align.dumb_consensus()
```

就行名字显示的那样，这是一个非常简单的保守序列计算器，它将只是在保守序列中累加每个位点的所有残基，如果最普遍的残基数大于某个阈值时，这个最普遍的残基将被添加到保守序列中。如果它没有到达这个阈值，将添加一个“不确定字符”。最终返回的保守序列对象是一个 Seq 对象，它的字母表是从组成保守序列所有序列的字母表中推断出来的。所以使用 print consensus 将给出如下信息：

```
consensus Seq('TATACATNAAAGNAGGGGATGCGGATAAATGGAAAGGCGAAAGAAAGAAAAAATGAAT
...', IUPACAmbiguousDNA())
```

你可以通过传入可选参数来调整 dumb_consensus 的工作方式：

the threshold 这是用来设定某个残基在某个位点出现的频率超过一定阈值，才将其添加到保守序列。默认为 0.7 (即 70%)。

the ambiguous character 指定保守序列中的不确定字符。默认为 'N'。

the consensus alphabet 指定保守序列的字母表。如果没有提供，我们将从比对序列的字母表基础上推断该字母表。

18.3.3 位点特异性评分矩阵

位点特异性评分矩阵 (Position specific score matrices, PSSMs) 以另一种总结比对信息的方式 (与刚才介绍的保守序列不同)，这或许在某些情况下更为有用。简单来说，PSSM 是一个计数矩阵。对于比对中的每一列，将所有可能出现的字母进行计数并相加。这些相加值将和一个代表序列 (默认为比对中的第一条序列) 一起显示出来。这个序列可能是保守序列，但也可以是比对中的任何序列。例如，对于比对，

```
GTATC
AT--C
CTGTC
```

它的 PSSM 是：

```
  G A T C
G 1 1 0 1
T 0 0 3 0
A 1 1 0 0
T 0 0 2 0
C 0 0 0 3
```

假设我们有一个比对对象叫做 `c_align`，为了获得 PSSM 和保守序列，我们首先得到一个摘要对象 (summary object)，并计算一致序列：

```
summary_align = AlignInfo.SummaryInfo(c_align)
consensus = summary_align.dumb_consensus()
```

现在，我们想创建 PSSM，但是在计算中忽略任何 N 不确定残基：

```
my_pssm = summary_align.pos_specific_score_matrix(consensus,
                                                  chars_to_ignore = ['N'])
```

关于此有亮点需要说明：

1. 为了维持字母表的严格性，你可以在 PSSM 的顶部显示比对对象字母表中规定的字符。空白字符 (Gaps) 并不包含在 PSSM 的顶轴中。
2. 传入并显示在左侧轴的序列可以不是保守序列。例如，你如果想要在 PSSM 左边显示比对中的第二条序列，你只需要：

```
second_seq = alignment.get_seq_by_num(1)
my_pssm = summary_align.pos_specific_score_matrix(second_seq,
                                                  chars_to_ignore = ['N'])
```

以上的命令将返回一个 PSSM 对象。为了显示出 PSSM，我们只需 `print my_pssm`，结果如下：

```

      A   C   G   T
T  0.0 0.0 0.0 7.0
A  7.0 0.0 0.0 0.0
T  0.0 0.0 0.0 7.0
A  7.0 0.0 0.0 0.0
C  0.0 7.0 0.0 0.0
A  7.0 0.0 0.0 0.0
T  0.0 0.0 0.0 7.0
T  1.0 0.0 0.0 6.0
...
```

你可以用 `your_pssm[sequence_number][residue_count.name]` 获得任何 PSSM 的元素。例如，获取上面 PSSM 中第二个元素的‘A’残基的计数，你可以：

```
>>> print my_pssm[1]["A"]
7.0
```

PSSM 类的结构有望使得获取元素和打印漂亮的矩阵都很方便。

18.3.4 信息量

一个潜在而有用的衡量进化保守性的测度是序列的信息量。

一个有用的针对分子生物学家的信息论的介绍可以在这里找到：<http://www.lecb.ncifcrf.gov/~toms/paper/primer/>。对于我们的目地，我们将查看保守序列或其部分序列的信息量。我们使用下面的公式计算多序列比对中某个特定的列的信息量：

$$IC_j = \sum_{i=1}^{N_a} P_{ij} \log \left(\frac{P_{ij}}{Q_i} \right) \quad (18.-2)$$

其中：

- IC_{j*} – 比对中第 j 列的信息量。
- N_{a*} – 字母表中字母的个数。
- P_{ij*} – 第 j 列的某个特定字母 i 的频率（即，如果 G 在包含有 6 个序列的比对中有 3 次出现，则该列 G 的信息量为 0.5）
- Q_{i*} – 字母 i 的期望频率。这是一个可选参数，由用户自行决定使用。默认情况下，它被自动赋值为 $0.05 = 1/20$ ，若为蛋白字母表；或 $0.25 = 1/4$ ，若为核酸字母表。这是在没有先验分布假设的情况下计算信息量。而在假设先验分布或使用非标准字母表时，你需要提供 Q_{i*} 的值。

好了，现在我们知道 Biopython 如何计算了序列比对的信息量，让我们看看怎么对部分比对区域进行计算。

首先，我们需要使用我们的比带来获得一个比对摘要对象，我们假设它叫做 `summary_align`（参见章节 18.3.1 来了解怎样得到它）。一旦我们得到这个对象，计算某个区域的信息量就像下面一样简单：

```
info_content = summary_align.information_content(5, 30,
                                                chars_to_ignore = ['N'])
```

哇哦，这比上面的公式看起来要简单多了！变量 `info_content` 现在含有一个浮点数来表示指定区域（比对中的 5 到 30）的信息量。我们在计算信息量时特意忽略了不确定残基 'N'，因为这个值没有包括在我们的字母表中（因而我们不必要关心它！）。

像上面提到的一样，我们同样能通过提供期望频率计算相对信息量：

```
expect_freq = {
    'A' : .3,
    'G' : .2,
    'T' : .3,
    'C' : .2}
```

期望值不能以原始的字典传入，而需要作为 `SubsMat.FreqTable` 对象传入（参见章节 20.2.2 以获得关于 `FreqTables` 的更多信息）。`FreqTable` 对象提供了一个关联字典和字母表的标准，这和 Biopython 中 `Seq` 类的工作方式类似。

要从频率字典创建一个 `FreqTable` 对象，你只需要：

```
from Bio.Alphabet import IUPAC
from Bio.SubsMat import FreqTable

e_freq_table = FreqTable.FreqTable(expect_freq, FreqTable.FREQ,
                                    IUPAC.unambiguous_dna)
```

现在我们得到了它，计算我们比对区域的相对信息量就像下面一样简单：

```
info_content = summary_align.information_content(5, 30,
                                                e_freq_table = e_freq_table,
                                                chars_to_ignore = ['N'])
```

现在，`info_content` 将包含与期望频率相关的该区域的相对信息量。

返回值是按上面的公式以 2 为对底数计算的。你可以通过传入 `log_base` 参数来改变成你想要的底数：

```
info_content = summary_align.information_content(5, 30, log_base = 10,
                                                chars_to_ignore = ['N'])
```

好了，现在你已经知道怎么计算信息量了。如果你要在实际的生命科学问题中应用它，最好找一些关于信息量的文献钻研，以了解它是怎样用的。希望你的钻研不会发现任何有关这个函数的编码错误。

18.4 替换矩阵

替换矩阵是每天生物信息学工作中的极端重要的一部分。它们提供决定两个不同的残基有多少相互替换的可能性的得分规则。这在序列比较中必不可少。Durbin 等的“Biological Sequence Analysis”一书中提供了对替换矩阵以及它们的用法的非常好的介绍。一些非常有名的替换矩阵是 PAM 和 BLOSUM 系列矩阵。

Biopython 提供了大量的常见替换矩阵，也提供了创建你自己的替换矩阵的功能。

18.4.1 使用常见替换矩阵

18.4.2 从序列比对创建你自己的替换矩阵

使用替换矩阵类能轻易做出的一个非常酷的事情，是从序列比对创建出你自己的替换矩阵。实际中，通常使用蛋白比对来做。在这个例子中，我们将首先得到一个 Biopython 比对对象，然后得到一个摘要对象来计算关于这个比对的相关信息。文件 `protein.aln`（也可在 [这里](#) 获取）包含 Clustalw 格式的比对输出。

```
>>> from Bio import AlignIO
>>> from Bio import Alphabet
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Align import AlignInfo
>>> filename = "protein.aln"
>>> alpha = Alphabet.Gapped(IUPAC.protein)
>>> c_align = AlignIO.read(filename, "clustal", alphabet=alpha)
>>> summary_align = AlignInfo.SummaryInfo(c_align)
```

章节 6.4.1 和 18.3.1 包含关于此类做法的更多信息。

现在我们得到了我们的 `summary_align` 对象，我们想使用它来找出不同的残基相互替换的次数。为了使例子的可读性更强，我们将只关注那些有极性电荷侧链的氨基酸。幸运的是，这能在生成替代字典时轻松实现，通过传入所有需要被忽略的字符。这样我们将能创建一个只包含带电荷的极性氨基酸的替代字典：

```
>>> replace_info = summary_align.replacement_dictionary(["G", "A", "V", "L", "I",
...                                                    "M", "P", "F", "W", "S",
...                                                    "T", "N", "Q", "Y", "C"])
```

这个关于氨基酸替代的信息以 python 字典的形式展示出来将会像如下的样子（顺序可能有所差异）：

```
{('R', 'R'): 2079.0, ('R', 'H'): 17.0, ('R', 'K'): 103.0, ('R', 'E'): 2.0,
 ('R', 'D'): 2.0, ('H', 'R'): 0, ('D', 'H'): 15.0, ('K', 'K'): 3218.0,
 ('K', 'H'): 24.0, ('H', 'K'): 8.0, ('E', 'H'): 15.0, ('H', 'H'): 1235.0,
 ('H', 'E'): 18.0, ('H', 'D'): 0, ('K', 'D'): 0, ('K', 'E'): 9.0,
 ('D', 'R'): 48.0, ('E', 'R'): 2.0, ('D', 'K'): 1.0, ('E', 'K'): 45.0,
 ('K', 'R'): 130.0, ('E', 'D'): 241.0, ('E', 'E'): 3305.0,
 ('D', 'E'): 270.0, ('D', 'D'): 2360.0}
```

这个信息提供了我们所需要的替换次数，或者说我们期望的不同的事情相互替换有多么频繁。事实上，（你可能会感到惊奇）这就是我们继续创建替代矩阵所需要的全部信息。首先，我们使用替代字典信息创建一个“接受替换矩阵”(Accepted Replacement Matrix, ARM)：

```
>>> from Bio import SubsMat
>>> my_arm = SubsMat.SeqMat(replace_info)
```

使用这个“接受替换矩阵”，我们能继续创建我们的对数矩阵（即一个标准类型的替换矩阵）：

```
>>> my_lom = SubsMat.make_log_odds_matrix(my_arm)
```

在创建的这个对数矩阵时有以下可选参数：

- `exp_freq_table` - 你可以传入一个每个字母的期望频率的表格。如果提供，在计算期望替换时，这将替代传入的“接收替换矩阵”。
- `logbase` - 用来创建对数奇数矩阵的对数底数。默认为 10。
- `factor` - 用来乘以每个矩阵元素的因数。默认为 10，这样通常可以使得矩阵的数据容易处理。
- `round_digit` - 矩阵中四舍五入所取的小数位数，默认为 0（即没有小数）。

一旦你获得了你的对数矩阵，你可以使用函数 `print_mat` 很漂亮的显示出来。使用我们创建的矩阵可以得到：

```
>>> my_lom.print_mat()
D   2
E  -1   1
H  -5  -4   3
K -10  -5  -4   1
R  -4  -8  -4  -2   2
    D   E   H   K   R
```

很好。我们现在得到了自己的替换矩阵！

18.5 BioSQL –存储序列到关系数据库中

BioSQL 是 OBF 多个项目（BioPerl、BioJava 等）为了支持共享的存储序列数据的数据库架构而共同努力的结果。理论上，你可以用 BioPerl 加载 GenBank 文件到数据库中，然后用 Biopython 从数据库中提取出来为一个包含 Feature 的 Record 对象——并获得或多或少和直接用 Bio.SeqIO（第 5 章）加载 GenBank 文件为 SeqRecord 相同的东西。

Biopython 中 BioSQL 模块的文档目前放在 <http://biopython.org/wiki/BioSQL>，是我们维基页面的一部分。

第 19 章 BIOPYTHON 测试框架

Biopython 具有一个基于 Python 标准单元测试框架 `unittest`<<http://docs.python.org/library/unittest.html>> 的回归测试框架 (文件 `run_tests.py`)。而为模块提供全面测试, 是确保 Biopython 代码在使用期内尽可能无 bug 的一个最重要的方面。也经常是最被轻视的方面之一。本章旨在使运行 Biopython 测试和编写测试代码尽可能容易。理想情况下, 进入 Biopython 的每个模块都应该有一个测试 (还应该对应文档!)。强烈鼓励我们所有开发者, 以及任何从源码安装 Biopython 的人运行单元测试。

19.1 运行测试

在你下载 Biopython 源码或者从我们的源码仓库签出时, 你会发现一个子目录调用 `Tests`。这包括关键脚本 `run_tests.py`、名为 `test_XXX.py` 的很多独立脚本、一个叫 `output` 的子目录和很多其他包含测试套件输入文件的子目录。

作为构建和安装 Biopython 的一部分, 你通常会在命令行上从 Biopython 源码顶层目录运行整个测试套件如下:

```
python setup.py test
```

这事实上等价于转到 `Tests` 子目录, 并运行:

```
python run_tests.py
```

你通常会想要只运行测试的一部分, 这可以如下来操作:

```
python run_tests.py test.SeqIO test.AlignIO.py
```

当给出测试列表时, `.py` 扩展名是可选的, 所以你可以只需打字:

```
python run_tests.py test.SeqIO test.AlignIO
```

要运行 docstring 测试 (见 19.3 节) 的话, 你可以用

```
python run_tests.py doctest
```

缺省情况下, `run_tests.py` 运行所有测试, 包括 docstring 测试。

如果一个测试失败了, 你还可以尝试直接运行它, 它会给出更多信息。

重要的是, 要注意单个单元测试有两类作用:

- 简单打印和比较脚本。这些单元测试本质上是简短的 Python 示例程序, 它们会打印出各种输出文本。对于一个名为 `test_XXX.py` 的测试文件, 在 `output` 子目录 (包含期望的输出) 下会有一个叫做 `test_XXX` 的匹配文本文件。测试框架所做的全部就是运行脚本并检查输出的一致性。

- 基于 `unittest` 的标准测试。这些会 `import unittest`，然后定义 `unittest.TestCase` 类，这些类的每一个都带有一或多个像以 `test_` 开头、检查代码的某些方面的方法那样的子测试。这些测试不应该直接打印任何输出。

目前，大约一半的 Biopython 测试是 `unittest` 风格的测试，另一半是 `print-and-compare` 测试。

直接运行一个简单的 `print-and-compare` 测试通常会在屏幕上给出大量输出，但是并不检查输出是否跟期望的一样。如果测试以一个意外的错误而失败，那么应该很容易精确定位脚本失败的位置。例如，对于一个 `print-and-compare` 测试，试一下：

```
python test.SeqIO.py
```

基于 `unittest` 的测试反倒是精确地显示你测试的哪一个小块失败了。例如，

```
python test.Cluster.py
```

19.2 编写测试

假如说你想为一个叫做 `Biospam` 的模块写一些测试。这可以是你写的一个模块，或者是一个还没有有任何测试的现存模块。在下面的例子中，我们假设 `Biospam` 是一个做简单数学的模块。

每个 Biopython 测试都可以有三个重要的文件和相关目录：

1. `test.Biospam.py` –关于你的模块的真正测试代码。
2. `Biospam [optional]` –一个包含任何必要输入文件的目录。任何会生成的输出文件也应该写在这里（并且最好在测试结束后打扫干净）以防堵塞主 `Tests` 目录。
3. `output/Biospam` – [只针对 `print-and-compare` 测试] 这个文件包括运行 `test.Biospam.py` 的期望输出。这个文件对于 `unittest` 风格的测试不是必须的，因为测试脚本 `test.Biospam.py` 会自己做验证。

你要自己决定你是想编写一个 `print-and-compare` 测试脚本还是一个 `unittest` 风格的测试脚本。重要的是你不能把这两种风格混合在一个测试脚本中。尤其是，不要在一个 `print-and-compare` 测试中使用“`unittest`”特征。

`Tests` 目录中任何具有 `test_` 前缀的脚本都会被 `run_tests.py` 找到并运行。下面，我们展示一个示例测试脚本 `test.Biospam.py`，针对一个 `print-and-compare` 测试和一个基于 `unittest` 的测试。如果你把这个脚本放进 Biopython 的 `Tests` 目录，那么 `run_tests.py` 就会找到它并执行其中包含的测试：

```
$ 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
```

19.2.1 编写一个 `print-and-compare` 测试

一个 `print-and-compare` 风格的测试对于初学者和新手来说是很容易写的 - 本质上它只是一个使用你的模块的示例脚本。

为了写一个关于 Biospam 的 print-and-compare 测试，这是你应该做的：

1. 编写一个叫 test.Biospam.py 的脚本

- 这个脚本应该位于 Tests 目录
- 脚本应该测试模块的所有重要功能（当然，你测试的越多、你的测试就越好！）。
- 尽量避免任何平台特异的东西，例如打印浮点数而不用显式格式字符串来避免有太多小数位（不同的平台会给出稍微不同的值）。

2. 如果脚本需要文件来进行测试，这些应转到目录 Tests/Biospam 中进行（如果你只需一些通用的东西，像一个 FASTA 序列文件，或者一条 GenBank 记录，试着用一个现存的样品输入文件来代替）。

3. 写出测试输出并验证输出是正确的。

有两种方法可以做到这一点：

- (a) 长期方法：

- 运行脚本并将输出写到一个文件中。在 UNIX（包括 Linux 和 Mac OS X）机器上，你可以这样做：`python test.Biospam.py > test.Biospam` 这会把输出写到文件 test.Biospam 中。
- 手动查看文件 test.Biospam 来确保输出正确。当你确定都没问题、没有 bug 后，你需要快速编辑 test.Biospam 文件使其首行为：`'test.Biospam'`（不含引号）。
- 复制文件 test.Biospam 到目录 Tests/output 中。

- (b) 快速方法：

- 运行 `python run_tests.py -g test.Biospam.py`。回归测试框架很聪明的会以他喜欢的方把输出放在恰当的地方。
 - 转到输出（应该在 Tests/output/test.Biospam）并复查输出以确保其完全正确。
4. 现在改换到 Tests 目录并运行 `python run_tests.py` 进行回归测试。这会运行所有测试，而你会看到你的测试也在运行（并通过）。
 5. 好了！这样你就得到了可用于签入或提交到 Biopython 的、关于你的模块的一个友好的测试。恭喜你！

例如，测试 Biospam 模块中的 addition 和 multiplication 功能的测试脚本 test.Biospam.py 也许看起来是下面这个样子：

```
from Bio import Biospam

print "2 + 3 =", Biospam.addition(2, 3)
print "9 - 1 =", Biospam.addition(9, -1)
print "2 * 3 =", Biospam.multiplication(2, 3)
print "9 * (- 1) =", Biospam.multiplication(9, -1)
```

我们用 `python run_tests.py -g test.Biospam.py` 来生成对应的输出，并检查输出文件 output/test.Biospam：

```
test_Biospam
2 + 3 = 5
9 - 1 = 8
2 * 3 = 6
9 * (- 1) = -9
```

通常，更大的 print-and-compare 测试的困难在于追踪输出行与测试脚本命令之间的对应关系。为此，打印出一些标记是很重要的，这些标记帮助你把输入脚本按行和产生的输出匹配起来。

19.2.2 编写一个基于 `unittest` 的测试

我们想要 Biopython 中的所有模块都具有单元测试，并且一个简单的 print-and-compare 测试比一点儿测试都没有要好。不过，尽管有一个陡峭的学习曲线，使用 `unittest` 框架能给出一个更结构化的结果，并且如果有一个测试失败，这能够清晰准确地指出测试的哪部分出了问题。子测试也可以单独运行，这对于测试和排错很有帮助。

从 2.1 版开始 `unittest` 框架就包含在 Python 中了，并且存档在 Python Library Reference（就是所推荐的你的枕边书）。也有 [关于 `unittest` 的在线文档](#)。如果你熟悉 `unittest` 系统（或类似于某些噪音测试框架的东西），你应该不会有什么麻烦。你也许发现，寻找 Biopython 中的现成例子很有帮助。

这是关于 Biospam 的一个 `unittest` 风格的极小测试脚本，你可以复制粘贴过去运行它：

```
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)
```

在分割测试中，我们使用 `assertAlmostEqual` 而不是 `assertEqual` 以免因舍入误差造成的测试失败；详情以及 `unittest` 中的其他可用功能参见 Python 文档中的 `unittest` 章节（[在线参考](#)）。

这里是基于 `unittest` 的测试的一些关键点：

- 测试实例存储在 `unittest.TestCase` 的子类中并涵盖了你的代码的一个基本方面。
- 对于任何在每个测试方法前后都要运行的重复代码，你可以使用方法 `setUp` 和 `tearDown`。例如 `setUp` 方法可用于创建你正在测试的对象的实例，或打开一个文件句柄。`tearDown` 可做任何整理，例如关闭文件句柄。
- 测试以 `test_` 为前缀并且每项测试应覆盖你所想要测试的内容的一个具体部分。一个类中你想包含多少个测试都行。
- 在测试脚本的末尾，你可以用

```
if __name__ == "__main__":
    runner = unittest.TextTestRunner(verbosity = 2)
    unittest.main(testRunner=runner)
```

来执行测试脚本，当脚本是从自己运行（而不是从 `run_tests.py` 导入）时。如果你运行该脚本，那么你会见到类似下面的东西：

```
$ 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
```

- 为了更清晰地表明每个测试都干了什么，你可以给每个测试加上 docstrings。它们会在运行测试的时候显示出来，如果一个测试失败这会是有用的信息。

```
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)
```

运行脚本你就会看到：

```
$ 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

如果你的模块包含 docstring 测试 (见19.3 小节), 你也许想在要运行的测试中包含这些。你可以修改 `if __name__ == "__main__":` 下面的代码如下面这样:

```
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)
```

这与你执行 `python test_Biospam.py` 时是否想要运行 docstring 测试有关; 运行 `python run_tests.py`, docstring 测试会自动运行 (假设他们被包含在 `run_tests.py` 中的 docstring 测试列表中, 见下面的小节)。

19.3 编写 doctests

Python 模块、类和函数支持使用 docstrings 创建文档。doctest 框架 (包含在 Python 中) 允许开发者将工作例子嵌入在 docstrings 中, 并自动测试这些例子。

目前只有一小部分 Biopython 包含 doctests。run_tests.py 脚本看护着 doctests 的运行。为此, run_tests.py 脚本开头是要测试的模块的一个手动编译列表, 该列表允许我们跳过那些可能没有安装可选外部依赖库的模块 (例如 Reportlab 和 NumPy 库)。所以如果你在 Biopython 模块中加一些针对 docstrings 的 doctests, 为了把它们包含在 Biopython 套件中, 你必须更新 run_tests.py 以包含你的模块。现在, run_tests.py 的相关部分看起来像下面这样:

```
# This is the list of modules containing docstring tests.
# If you develop docstring tests for other modules, please add
# those modules here.
DOCTEST_MODULES = ["Bio.Seq",
                   "Bio.SeqRecord",
                   "Bio.SeqIO",
                   "...",
                   ]
#Silently ignore any doctests for modules requiring numpy!
try:
    import numpy
    DOCTEST_MODULES.extend(["Bio.Statistics.lowess"])
except ImportError:
    pass
```

注意我们首先把 doctests 看做文档, 所以你应该坚持典型用法。通常处理错误条件等诸如此类的复杂例子最好留给一个专门的单元测试。

注意, 如果你想编写涉及文件解析的 doctests, 定义文件位置复杂性是很要紧的。理想情况下, 假设代码会从 Tests 目录运行, 使用相对路径即可, 关于这一点的一个例子参见 Bio.SeqIO doctests。

要想只运行 docstring 测试, 使用

```
$ python run_tests.py doctest
```

第 20 章高级

20.1 解析器的设计

过去很多 Biopython 解析器都是根据面向事件设计出来的，包括 Scanner 和 Consumer。

Scanners 是将输入的数据源进行逐行分析，只要识别出数据中的信息就会发送一个事件。例如，如果数据中包含物种名信息，Scanner 只要读到某行包含名称信息时就会产生一个 `organism_name` 事件。

Consumers 是用来接收 Scanners 所发出事件的对象。接着上面的例子，当 Consumer 收到了 `organism_name` 事件，在当前应用程序中无论以何种方式都会运行。

这是一个非常灵活的构架，如果你想要将一个文件解析成多种其他格式的，这将会很有优势。例如，Bio.GenBank 模块可以运用这种方式构建 SeqRecord 或者其他独特的文件格式记录对象。

最近，很多添加了 Bio.SeqIO 和 Bio.AlignIO 的解析器使用了一种更为简单的方法，但是只能产生单一形式的文件格式（分别是 SeqRecord and MultipleSeqAlignment）。在某些情况，Bio.SeqIO 解析器实际上包含了另一种 Biopython 解析器 - 例如，Bio.SwissProt 解析器产生了特定的 SwissProt 格式对象，又转换成了 SeqRecord 格式对象。

20.2 替换矩阵

20.2.1 SubsMat

这个模块提供了一个类和一些固定的方法来产生替换矩阵，类似于 BLOSUM 或者 PAM 矩阵，但是是基于用户提供的数据。此外，你还可以从已建立的替换矩阵集合 MatrixInfo.py 中选择一个矩阵。SeqMat 类来自于一个字典（dictionary）：

```
class SeqMat(dict)
```

这个字典的格式是 $\{(i1,j1):n1, (i1,j2):n2, \dots, (ik,jk):nk\}$ ， i 和 j 是字母集，而 n 是一个值。

1. 属性

- (a) `self.alphabet`: Bio.Alphabet 中定义的一个类
- (b) `self.ab_list`: 排列好的字母列表。主要是内部需求。

2. 方法

- (a) `__init__(self,data=None,alphabet=None, mat_name='', build_later=0):`

- i. `data`: 可以是一个字典，也可以是另一个 SeqMat 实例。
- ii. `alphabet`: 一个 Bio.Alphabet 的实例。如果没有提供，将从数据构建一个 alphabet。

iii. `mat_name`: 矩阵名, 例如 BLOSUM62 或者 PAM250

iv. `build_later`: 默认值为 `false`。如果为 `true`, 用户应该只提供 `alphabet` 和空字典。如果想要之后再构建矩阵, 这样会跳过 `alphabet` 大小和矩阵大小的检查。

(b) `entropy(self, obs_freq_mat)`

i. `obs_freq_mat`: 一个观测频率矩阵。基于“`obs_freq_mat`”的频率返回矩阵的熵值。矩阵实例须为 LO 或者 SUBS。

(c) `sum(self)`

计算矩阵的字母表中每个字母值的总和, 返回值是字典的形式 `{i1: s1, i2: s2, ..., in: sn}`, 其中:

- `i`: 一个字母;
- `s`: 半矩阵中某个字母值的总和;
- `n`: `alphabet` 中字母的个数。

(d) `print_mat(self, f, format="%4d", bottomformat="%4s", alphabet=None)`

将矩阵打印到文件句柄 `f`。 `format` 是矩阵值的格式; `bottomformat` 是底部行 (矩阵字母) 的格式。下面是一个 3 字母矩阵的例子:

```
A 23
B 12 34
C 7 22 27
  A  B  C
```

`alphabet` 可选参数是 `alphabet` 中所有字符的一个字符串。如果用户提供了数据, 则轴上字母的顺序是根据字符串中的顺序, 而不是字母表的顺序。

3. 用法

安排下面这部分是因为大多数读者希望知道如何产生一个对数机率矩阵 (log-odds matrix)。当然, 也可以生成和研究过渡矩阵 (log-odds matrix)。但是大部分的人只是想要一个对数机率矩阵, 仅此而已。

(a) 产生一个可接受的替代矩阵

首先, 你应该从数据中产生出一个可接受替代矩阵 (accepted replacement matrix, ARM)。ARM 中的数值是根据你的数据中替换的个数决定的。数据可以是一对或者多对的序列比对结果。例如, 丙氨酸被半胱氨酸替换了 10 次, 而半胱氨酸被丙氨酸替换了 12 次, 其相对应的 ARM 为:

```
('A','C'): 10, ('C','A'): 12
```

由于顺序并不重要, 用户也可以只用一个输入:

```
('A','C'): 22
```

一个 SeqMat 实例的初始化可以用全矩阵 (第一种计数方法: 10,12), 也可以用半矩阵 (后一种方法, 22)。一个蛋白字母全矩阵的大小应该是 $20 \times 20 = 400$ 。而一个这样的半矩阵大小是 $20 \times 20 / 2 + 20 / 2 = 210$ 。这是因为相同字母的输入并没有改变 (矩阵的对角线)。如果一个大小为 `N` 的 `alphabet`:

- i. 全矩阵大小: $N \times N$
- ii. 半矩阵大小: $N(N+1)/2$

如果传递的是全矩阵，SeqMat 的构造函数会自动产生半矩阵。如果传递的是半矩阵，则键的字母将按照字母表顺序排列 ('A','C')，而不是 ('C','A')。

讲到这里，如果你想知道的仅仅只是怎样产生一个对数机率矩阵的话，请直接看用法示例那个章节。对于想要更加深入地知道核苷酸/氨基酸频率数据的读者，接下来要讲的是内部函数的细节。

(b) 生成观测频率矩阵 (observed frequency matrix, OFM)

用法:

```
OFM = SubsMat._build_obs_freq_mat(ARM)
```

OFM 是由 ARM 产生的，只是将替换的个数换成了替换频率。

(c) 生成期望频率矩阵 (expected frequency matrix, EFM)

用法:

```
EFM = SubsMat._build_exp_freq_mat(OFM,exp_freq_table)
```

- i. `exp_freq_table`: 为一个 `FreqTable` 的实例。有关 `FreqTable` 更多信息请见第 20.2.2 节。简单地说，期望频率表表示字母表中每个元素显示的频率。这个表相当于一个字典，字母是键，字母对应的频率是值。总和为 1。

期望频率表可以（理论上说也应该可以）从 OFM 得到。所以大多数情况你可以用下面的代码产生 `exp_freq_table`:

```
>>> exp_freq_table = SubsMat._exp_freq_table_from_obs_freq(OFM)
>>> EFM = SubsMat._build_exp_freq_mat(OFM,exp_freq_table)
```

如果需要，你也可以使用自己提供的 `exp_freq_table`。

(d) 生成替换频率矩阵 (substitution frequency matrix, SFM)

用法:

```
SFM = SubsMat._build_subs_mat(OFM,EFM)
```

使用观察频率矩阵 (OFM) 和期望频率矩阵 (EFM)，得到相应值的除法结果。

(e) 生成对数机率矩阵 (log-odds matrix, LOM)

用法:

```
LOM=SubsMat._build_log_odds_mat(SFM[,logbase=10,factor=10.0,round.digit=1])
```

- i. 使用替换频率矩阵 (SFM)。
- ii. `logbase`: 用来产生对数机率值的对数的底。
- iii. `factor`: 对数机率值的乘数因子。每个数通过 `log(LOM[key])*factor` 产生，如果需要，还可以四舍五入到 `round.digit` 指定的小数点位数。

4. 用法示例

因为大部分人都想用最简单的方法产生对数机率矩阵 (LOM)，SubsMat 提供了一个可以完成所有需求的函数：

```
make_log_odds_matrix(acc_rep_mat,exp_freq_table=None,logbase=10,
                    factor=10.0,round.digit=0):
```

(a) `acc_rep_mat`: 用户提供可接受替代矩阵 (ARM)

(b) `exp_freq_table`: 期望频率表。如果用户没有提供，就从 `acc_rep_mat` 产生。

- (c) logbase: LOM 的对数的底。默认为 10。
- (d) round_digit: 四舍五入的小数点位数。默认为 0。

20.2.2 FreqTable

`FreqTable.FreqTable(UserDict.UserDict)`

1. 属性:

- (a) alphabet: 一个 `Bio.Alphabet` 实例。
- (b) data: 频率字典
- (c) count: 计数字典 (如果有计数的话)。

2. 功能:

- (a) `read_count(f)`: 从 `f` 读入一个计数文件。然后将其转换成频率。
- (b) `read_freq(f)`: 从 `f` 读入一个频率数据文件。当然, 我们不用计数, 我们感兴趣的是字母频率。

3. 用法示例: 文件中有残基的个数, 用空格分格, 形式如下 (以 3 个字母为例):

```
A   35
B   65
C  100
```

用 `FreqTable.read_count(file_handle)` 函数读入。

一个等价的频率文件:

```
A  0.175
B  0.325
C  0.5
```

反之, 残基频率或者计数也可以作为字典输入。一个计数字典的例子 (3 个字母):

```
{'A': 35, 'B': 65, 'C': 100}
```

这也意味着 'C' 的频率是 0.5, 'B' 的频率是 0.325, 'A' 的频率是 0.175。A、B、C 的总和为 200。

一个相同数据的频率字典如下:

```
{'A': 0.175, 'B': 0.325, 'C': 0.5}
```

总和为 1。

当传入一个字典数据作为参数, 应该指出这是一个计数还是频率的字典。因此 `FreqTable` 类的构造函数需要两个参数: 字典本身和 `FreqTable.COUNT` 或者 `FreqTable.FREQ`, 分别代表计数或者频率。

读入期望的计数, `readCount` 会产生频率。下面的任意一个都可以用来产生频率表 (ftab):

```
>>> from SubsMat import *
>>> ftab = FreqTable.FreqTable(my_frequency_dictionary, FreqTable.FREQ)
>>> ftab = FreqTable.FreqTable(my_count_dictionary, FreqTable.COUNT)
>>> ftab = FreqTable.read_count(open('myCountFile'))
>>> ftab = FreqTable.read_frequency(open('myFrequencyFile'))
```

第 21 章为 BIOPYTHON 做贡献

21.1 错误报告与功能需求

对于 biopython 的开发者来说，从使用者得到反馈是非常重要的；来自于众多使用者的问题反馈、错误报告（以及补丁）对于 biopython 这样的开源项目有着莫大的帮助。

Biopython 邮件列表 是一个讨论功能需求和潜在 bugs 的主要论坛：

- biopython@biopython.org -讨论任何跟 biopython 有关的问题。
- biopython-dev@biopython.org -开发者用于讨论 biopython 开发的邮件列表（其他人员同样可以做出贡献）

除此之外，如果你发现 bug，请提交到我们的 bug 监控页面 (<http://redmine.open-bio.org/projects/biopython>) 这样，这些 bug 才不至于淹没于浩如烟海的收件箱中被遗忘。

21.2 邮件列表与帮助新手

我们希望所有的 biopython 使用者都在 biopython 主邮件列表注册并订阅相关邮件。同时使用者在对某一领域有一定了解之后，也希望他们能够帮助入门者并对他们的问题进行解答，毕竟每个人开始的时候都是菜鸟。

21.3 贡献文档

我们很高兴能从用户那里收到反馈，不论是通过 bug 报告还是邮件列表的形式。或许在你阅读这一教程的时候，你可能会发现某些你所感兴趣的话题没被涉及或者没有解释清楚。甚至 biopython 的内建文档 (docstrings) 以及一些在线文档 (<http://biopython.org/DIST/docs/api>) 有所缺失，有能力的用户都可以帮助我们填补这些空白。

21.4 贡献学习手册示例

如 18 章中用户手册解释的那样，如今 biopython 有一个汇集用户贡献的使用文档的 wiki 文库 (<http://biopython.org/wiki/Category:Cookbook>)，也许你也可以把做的一些使用说明放在这里。

21.5 维护跨平台发行版本

目前我们提供 biopython 的源代码安装（适用于任何操作系统，如果你安装了正确的编译工具的话），同时还提供有点击运行的 windows 安装工具。这涵盖了当今大多数操作系统。

对于大多数 linux 操作系统来说，都有志愿者将源代码进行编译成包，同时也解决了包依赖等相关问题，linux 用户可以很容易的安装。对于这些工作，我们非常感激。如果你想要对于这一工作有所贡献的话，请阅读你的 linux 发行版关于这一事项的详细说明。

对于想要从事这一工作的用户，以下是一些关于主要操作系统平台的建议：

Windows – Windows 产品通常有一个图形安装工具，能够将 biopython 所有的基本组建安装到相应的目录。我们使用 Distutils 工具来创建能完成这一任务的 installer。

首先，你要确定你的 Windows 操作系统中安装了 C 编译器，而且你能够正确编译并安装（通常这是困难的地方），可以参考 Biopython 安装说明。

在你配置好 C 编译器之后，安装工作就很简单：

```
python setup.py bdist.wininst
```

目前，在 32 位 Windows 操作系统上安装 biopython 没有任何问题。只是 64 位操作系统的安装还没能实现，要是谁能在这方面做点贡献就太好了。

RPMs – RPMs 在一些 linux 发行版中是非常流行的包管理系统，在 RPMs 主页 (<http://www.rpm.org>) 有很多关于 RPMs 的文档说明帮助你快速的开始。为你的系统创建一个 RMP 是很容易的一件事。你仅仅需要将源代码编译成包（需要一个能正常工作的 C 编译器），更多信息可以参考 Biopython 安装说明。

创建一个 RPM, 仅需做以下工作：

```
python setup.py bdist.rpm
```

在终端运行以上命令后，会在 dist 目录下创建一个跟你使用的操作系统相关的 RPM，以及一个 RPM 源文件。一般情况下，这个 RPM 能正常工作。简洁而实用！

Macintosh –由于 Apple 迁移到 Mac OS X, 很多工作就简单多了。一般来说，Mac 操作系统相当于一个 Unix 变体，Biopython 源码的安装同在 linux 上一样容易。安装 Apple's X-code 是安装所有 GCC 编译器最简易的方法。我们或许会为 Mac 操作系统提供点击-安装的工具，但目前为止还没必要。

一旦得到一个安装包，务必在你的系统上检测并确保所有文件能够正确安装并能正常工作。在这一工作完成之后，你的工作就结束了，剩下的就是 Biopython 开发者的事了（如果不确定该发送给谁，可以给 Biopython 主邮件列表发送邮件），谢谢！

21.6 贡献单元测试 (Unit Tests)

即使你没有任何新的功能添加到 Biopython，但仍然想写一些代码，请考虑增加我们的单元测试。我们整个第 19 章都在讲这个内容。

21.7 贡献源码

除了使用 Python 语言开发生物学相关的程序外，任何人都可以没有限制地加入 Biopython 源码的开发。任何人若对某方面的编程感兴趣，Biopython 邮件列表是讨论此事最合适的地方——只需告知我们你的兴趣所在或工作内容。通常来讲，在开发某个模块之前，我们会在邮件列表里讨论此事，因为这样做会有助于产生好的想法，讨论完成之后，就剩下编程了！

主要的 Biopython 发布版本会尽量做到统一和通用，以方便用户的使用。在附带文档 (<http://biopython.org/wiki/Contributing>) 中，你可以获取在 Biopython 中用到的编程方式的原则。同时，我们也尽量在发行版和文档中加入源码和测试（关于 regression testing framework 详见19章），以使得各方面能保持一致并正常工作。

值得注意的是，你需要有合法的权利去贡献源码并且在 Biopython 发行许可下发布。当然了，要是你的程序完全是由你自己编写，没有任何其他的代码，就不要为此担心了。另外，在贡献衍生版本的时候，会有些问题——比如说一些给予 GPL 或者 LGPL 的程序与 Biopython 许可不相容。如果你有什么疑问，请在 biopython-dev 邮件列表里讨论。

另外一个关于向 Biopython 贡献源码的问题涉及到开发和运行时依赖问题。一般来讲，编写程序调用像 BLAST、EMBOSS 或者 ClustalW 这样的独立程序没什么问题。但是，任何依赖于其他文库的程序——即使是 Python 文库（尤其是像 NumPy 这样用于编译和安装 Biopython 的文库）就需要做进一步的讨论。

除此之外，如果你手头有某些代码，而你又觉得不适合发行版，却又想共享出来，你可以将它们放在一个专门收集生物信息学 Python 代码的地方 (<http://biopython.org/wiki/Scriptcentral>)，

希望这个文档能在使用 Biopython 的过程中带给你想要的信息，当然了，最重要的就是贡献。

第 22 章附录：PYTHON 相关知识

如果你对 Python 编程还不熟练，那么你在使用 Biopython 过程中遇到的问题通常与 Python 本身有关。本章节主要向读者提供一些使用 Biopython 文库过程中有用的建议和一些常用代码（至少对我们来说很常用）。关于这一章的内容，你要是有什么好的建议的话，请告诉我们。

22.1 到底什么是句柄 (handle)？

这个文档中，句柄常被提及，而且也比较难理解（至少对我来说）。一般说来，你可以把句柄想象成一个对文本信息的“封装”。

相比普通文本信息，使用句柄至少有两个好处：

1. 对于以不同方式存储的信息，句柄提供了一个标准的处理方法。这些文本信息可能来自文件、内存中的一个字符串、命令行指令的输出或者来自于远程网站信息，但是句柄提供了一种通用的方式处理这些不同格式和来源的文本信息。
2. 句柄可以依次读取文本信息，而不是一次读取所有信息。这点在处理超大文件时尤为有用，因为一次载入一个大文件可能会占去你所有的内存。

不论是从文件读取文本信息还是将文本信息写入文件，句柄都能胜任。在读取文件时，常用的函数有 `read()` 和 `readline()`，前者可以通过句柄读取所有文本信息，而后者则每次读取一行；对于文本信息的写入，则通常使用 `write()` 函数。

句柄最常见的使用就是从文件读取信息，这可以通过 Python 内置函数 `open` 来完成。下面示例中，我们打开一个指向文件 `m_cold.fasta`（可通过网址 http://biopython.org/DIST/docs/tutorial/examples/m_cold.fasta 获取）的句柄：

```
>>> handle = open("m_cold.fasta", "r")
>>> handle.readline()
">gi|8332116|gb|BE037100.1|BE037100 MP14H09 MP Mesembryanthemum ...\\n"
```

Biopython 中句柄常用来向解析器 (parsers) 传递信息。比如说，自 Biopython1.54 版本后，`Bio.SeqIO` 和 `Bio.AlignIO` 模块中的主要函数都可以使用文件名来代替句柄使用：

```
from Bio import SeqIO
for record in SeqIO.parse("m_cold.fasta", "fasta"):
    print record.id, len(record)
```

在比较早的 BioPython 版本中，必须使用句柄：

```
from Bio import SeqIO
handle = open("m_cold.fasta", "r")
for record in SeqIO.parse(handle, "fasta"):
    print record.id, len(record)
handle.close()
```

这种操作方式仍有其用武之地，比如在解析一个 gzip 压缩的 FASTA 文件中：

```
import gzip
from Bio import SeqIO
handle = gzip.open("m_cold.fasta.gz")
for record in SeqIO.parse(handle, "fasta"):
    print record.id, len(record)
handle.close()
```

在第5.2节中有更多此类示例可供参考，其中包括 bzip2 压缩文件的读取。

22.1.1 从字符串创建句柄

一个比较有用的工具是将字符串中包含的文本信息传递给一个句柄。以下示例是使用 Python 标准文库 cStringIO 来展示如何实现的：

```
>>> my_info = 'A string\n with multiple lines.'
>>> print my_info
A string
  with multiple lines.
>>> from StringIO import StringIO
>>> my_info_handle = StringIO(my_info)
>>> first_line = my_info_handle.readline()
>>> print first_line
A string

>>> second_line = my_info_handle.readline()
>>> print second_line
  with multiple lines.
```


REFERENCES

- [1] 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). doi:10.1093/bioinformatics/btp163,
- [2] 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). doi:10.1093/bioinformatics/btk021,
- [3] 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). doi:10.1146/annurev.phyto.44.070505.143444,
- [4] Gldine A. van der Auwera, Jaroslaw E. Kraruo 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). doi:10.1007/s10482-009-9316-9
- [5] Caroline Proux, Douwe van Sinderen, Juan Suarez, Pilar Garcia, Victor Ladero, Gerald F. Fitzgerald, Frank Desiere, Harald Brssow: “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). <http://dx.doi.org/10.1128/JB.184.21.6026-6036.2002>
- [6] Florian Jupe, Leighton Pritchard, Graham J. Etherington, Katrin MacKenzie, Peter JA Cock, Frank Wright, Sanjeev Kumar Sharmal, 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). <http://dx.doi.org/10.1186/1471-2164-13-75>
- [7] 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). doi:10.1093/nar/gkp1137
- [8] Patrick O. Brown, David Botstein: “Exploring the new world of the genome with DNA microarrays”. *Nature Genetics* **21** (Supplement 1), 33 – 37 (1999). doi:10.1038/4462
- [9] 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). doi:10.1186/1471-2105-13-209
- [10] Athel Cornish-Bowden: “Nomenclature for incompletely specified bases in nucleic acid sequences: Recommendations 1984.” *Nucleic Acids Research* **13** (9): 3021 – 3030 (1985). doi:10.1093/nar/13.9.3021
- [11] Douglas R. Cavener: “Comparison of the consensus sequence flanking translational start sites in *Drosophila* and vertebrates.” *Nucleic Acids Research* **15** (4): 1353 – 1361 (1987). doi:10.1093/nar/15.4.1353

- [12] 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).
- [13] Brad Chapman and Jeff Chang: “Biopython: Python tools for computational biology”. *ACM SIGBIO Newsletter* **20** (2): 15 – 19 (August 2000).
- [14] Michiel J. L. de Hoon, Seiya Imoto, John Nolan, Satoru Miyano: “Open source clustering software”. *Bioinformatics* **20** (9): 1453 – 1454 (2004). doi:10.1093/bioinformatics/bth078
- [15] 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 Science USA* **95** (25): 14863 – 14868 (1998). doi:10.1073/pnas.96.19.10943-c
- [16] 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).
- [17] Gene H. Golub, Charles F. Van Loan: *Matrix computations*, 2nd edition (1989).
- [18] Thomas Hamelryck and Bernard Manderick: 11PDB parser and structure class implemented in Python” . *Bioinformatics*, **19** (17): 2308 – 2310 (2003) doi: 10.1093/bioinformatics/btg299.
- [19] Thomas Hamelryck: “Efficient identification of side-chain patterns using a multidimensional index tree” . *Proteins* **51** (1): 96 – 108 (2003). doi:10.1002/prot.10338
- [20] 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). doi:10.1002/prot.20379.
- [21] John A. Hartiga. *Clustering algorithms*. New York: Wiley (1975).
- [22] Anil L. Jain, Richard C. Dubes: *Algorithms for clustering data*. Englewood Cliffs, N.J.: Prentice Hall (1988).
- [23] Voratas Kachitvichyanukul, Bruce W. Schmeiser: Binomial Random Variate Generation. *Communications of the ACM* **31** (2): 216 – 222 (1988). doi:10.1145/42372.42381
- [24] Teuvo Kohonen: “Self-organizing maps”, 2nd Edition. Berlin; New York: Springer-Verlag (1997).
- [25] Pierre L’Ecuyer: “Efficient and Portable Combined Random Number Generators.” *Communications of the ACM* **31** (6): 742 – 749,774 (1988). doi:10.1145/62959.62969
- [26] 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). doi:10.1186/1471-2105-6-202.
- [27] V. Matys, E. Fricke, R. Geffers, E. G?ssling, 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. Mnch, 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). doi:10.1093/nar/gkg108
- [28] Robin Sibson: “SLINK: An optimally efficient algorithm for the single-link cluster method”. *The Computer Journal* **16** (1): 30 – 34 (1973). doi:10.1093/comjnl/16.1.30
- [29] George W. Snedecor, William G. Cochran: *Statistical methods*. Ames, Iowa: Iowa State University Press (1989).
- [30] 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 Science USA* **96** (6): 2907 – 2912 (1999). doi:10.1073/pnas.96.6.2907
- [31] Robert C. Tryon, Daniel E. Bailey: *Cluster analysis*. New York: McGraw-Hill (1970).

- [32] John W. Tukey: “Exploratory data analysis”. Reading, Mass.: Addison-Wesley Pub. Co. (1977).
- [33] Ka Yee Yeung, Walter L. Ruzzo: “Principal Component Analysis for clustering gene expression data”. *Bioinformatics* **17** (9): 763 – 774 (2001). doi:10.1093/bioinformatics/17.9.763
- [34] Alok Saldanha: “Java Treeview—extensible visualization of microarray data”. *Bioinformatics* **20** (17): 3246 – 3248 (2004). <http://dx.doi.org/10.1093/bioinformatics/bth349>

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*