BIOPYTHON

中文指南

生物信息学论坛 2010 倾情奉献

http://www.bioxxx.cn

----------2010 年第一版----------

生物信息学论坛工作

目录

前言

本指南是生物信息学论坛（http://www.bioxxx.cn）根据网页 http://zonghe.17xie.com/book/10781126/的对英文翻译的整理，以方便大家对指南的查阅，在整理过程中难免有疏漏，望广大读者的使用过程中能将错误及时反馈给本站管理员。本人将及时更新。

祝大家使用愉快！

联系方式

邮箱：webmaster@bioxxx.cn

QQ：466628669

生物信息学论坛

2010-1-1

第一章 介绍

1.1 什么是 biopython？

Biopython 计划是一个使用 python 来开发计算分子生物学工具的国际社团(http://www.python.org/). 该网址 http://www.biopython.org/提供一个在线的基于 python 的生命科学研究的模块，脚本和网络链接。基本来说，我们喜欢使用 python 来编程，并且希望对生物信息学来说尽量容易的使用 python 创建高质量，可重用的模块和脚本。

1.1.1 在 biopython 工具包中能够发现什么

主要的 Biopython 发行版有很多功能,包括:

将生物信息学文件分析成 python 可利用的数据结构, 包括以下支持的格式

o Blast 输出 – standalone 和网络 blast

o Clustalw

o FASTA

o GenBank

o PubMed 和 Medline

o Expasy 文件, 例如 Enzyme, Prodoc 和 Prosite

o SCOP, 包括‘dom’和‘lin’文件

o UniGene

o 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。

处理常用的在线生物信息学数据库代码：

o NCBI – Blast, Entrez 和 PubMed 服务

o Expasy – Prodoc 和 Prosite 条目

常用生物信息程序的界面，例如：

o Standalone Blast from NCBI

o Clustalw 比对程序

? 一个标准序列类处理序列、ID 和序列特征

? 对序列处理的常规操作，例如翻译，转录和重量计算。

? 代码以处理使用 k 最近邻接，Bayes 或 SVM 进行分类

? 代码以处理比对，包括标准方法和处理替换矩阵

? 代码以轻易分解平行任务为分离的过程
? GUI 程序以进行基础的序列操作，翻译，BLAST 等
? 使用模块的扩展文档和帮助，包括这个文件，在线 wiki 文档和网站以及 mail 列表
? 和其他语言进行整合，包括 Bioperl 和 Biojava 计划, 使用 BioCorba 接口标准.
我们希望这给了你足够的原因下载和开始使用 Biopython!

1.2 安装 biopython

首先要安装 python，然后选择对应的 biopython 包进行安装即可。windows 下建议选择 2.5 版本。

1.3 FAQ

1，为什么 Bio.SeqIO 不工作?载入文件但是没有分析函数。
这是因为旧版本中不包括一些相关的代码，需要用 1.43 以上的

2. Why doesn't Bio.SeqIO.read() work?

The module imports fine but there is no read function!需要 Biopython 1.45 以上

3. Why doesn't Bio.Blast work with the latest plain text NCBI blast output?

The NCBI keep tweaking the plain text output from the BLAST tools,和 keeping our parser up to date is an ongoing struggle. We recommend you use the XML output instead, which is designed to be read by a computer program.

4. Why isn't Bio.AlignIO) present? The module import fails!

You need Biopython 1.46 or later.

5. Why doesn't Bio.Entrez.read() work? The module imports fine but there is no read function!

You need Biopython 1.46 or later.

6. I looked in a directory for code, but I couldn't seem to find the code that does something. Where's it hidden?

One thing to know is that we put code in __init__.py files. If you are not used to looking for code in this file this can be confusing. The reason we do this is to make the imports easier for users. For instance, instead of having to do a "repetitive" import like from Bio.GenBank import GenBank, you can just import like from Bio import GenBank.

第二章 快速开始-可以用 biopython 来做什么？

这一节被设计成为使你可以快速开始 biopython，给你一个什么是可用的概览及如何使用它.所有的例子都假定你有一些基本的 python 知识，并且已经成功安装上 biopython。如果你希望复习 python，你可用从 python 官网开始，那里提供了免费的文档。因为很多的生物工作需要联网，因此一些例子同样需要联网才能运行。现在开始吧！

2.1 General overview of what Biopython provides

正如在介绍里讲的, Biopython 提供了生物学家使用计算机处理他们所关心问题的一个库. 因此，这意味着你需要有些 python 的编程经验，至少有这方面的兴趣. Biopython 通过给程序员提供可重用的库以使你的工作更容易，这样你就可以集中精力回答你特定的问题，而不是专注于分析一个特定文件格式的内部结构。（当然，如果你想帮助写一个并不存在的分析器，同时想对 biopython 有贡献的话，尽管去做--即专注编程）故 Biopython 的工作使让你更快乐!

需要指出的是，biopython 往往提供了多种处理“相同问题”的方法。对我来说，这有些令人烦恼，因为我总是使用最正确的方法来完成某个任务。但是，这也是一个优点，因为它给了你很多弹性和对库的控制。这个指南给你提供了常用的或简单的方式来处理问题。这样你就可以做事情了。想要学更多处理问题的可能性方法，查看 cookbook 部分（给了你很多酷的技巧和贴士）和高级部分（提供了很多你想知道的细节）

2.2 处理序列

当然，生物信息学中最重要的对象就是序列。因此，我们以 biopython 处理序列技巧开始介绍，Seq 对象,在第三章会详细介绍。通常当我们思考序列时，我们的想法是像'AGTACACTGGT'这样的一串字母。你可以像下面一样，创建这样的 Seq 对象。“>>>” 是 python 的提示符，我们在其后输入：

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
```

```
>>> my_seq.alphabet
Alphabet()
>>> print my_seq.tostring()
AGTACACTGGT
```

我们这里的序列对象有一个一般的字母表--我们不确定它是 DNA 或者蛋白质序列（包含 AGTC 的蛋白质序列）。我们将在第三章更多的讨论字母表。加入字母表后，Seq 对象就不同于 python 支持的方法中的字符串。你不能使用一个简单的字符串。

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

下一个最重要的类是 SeqRecord 或序列记录。这个类包含一个序列（作为 Seq 对象），带有附加的注释，包括标识符，名字和描述。读取和写入序列文件格式的 Bio.SeqIO 模块和 SeqRecord 对象一起工作，将在下面介绍，并且将详细的在第四章介绍。这包含了 Biopython 序列类的基本特性和使用。既然我们已经对 Biopython 库有了一定的了解，就可以深入研究处理生物学文件格式的美妙世界了。

2.3 一个使用的例子

在我们急于开始分析和用 Biopython 处理事物之前，让我们举一个例子以激发我们所做的一切，同时使生活更有趣。毕竟，如果没有一些生物学在这个简介中，你为什么还要读它呢？因为我喜欢植物，我想我们还是以一个植物的例子开始（向那些喜欢其他物种的朋友致歉）。最近从我们当地的花房回来后，我们对 Lady Slipper Orchids（一种兰花）有一种难以想象的困惑（如果你想知道为什么，看看它们的图片就知道了）。当然，兰花并不仅仅好看，它们对那些研究进化和分类的人来说也是非常有趣的。

因此，让我们假定我们想写一个基金提议对它的进化做下分子研究，看看已有的研究，同时看我们能否在加些进入。经过一些钻研后，我们发现 Lady Slipper

Orchids 是兰花科，兰花亚科，由 5 个属组成：Cypripedium, Paphiopedilum, Phragmipedium, Selenipedium 以及 Mexipedium.

那已经给了我们足够的信息以获得更多的信息。因此，让我们看看 Biopython 工具如何帮助我们。我们将在 2.4 节开始序列分析。但是，兰花会过后回来。例如，在第八章，我们从 SwissProt 获得兰花蛋白质，在 9.1 节中搜索 PubMed 以获得兰花相关的文献，在 9.2.1 节中从 GenBank 中提取兰花的序列数据，在 9.3.1 节中对兰花蛋白质进行 ClustalW 多序列比对。

2.4 分析序列文件格式

生物信息学工作的很大一部分涉及到处理存储着数据的不同文件格式。这些文件存有感兴趣的生物学数据，一个特别的挑战就是将这些文件分析成一种格式，这样你就可以使用某种编程语言来操控它们。尽管分析这些文件的工作会遇到挫折，因为格式可以很有规律的改变，那种格式可能包含细微的差别，它也许会破坏甚至是设计最好的分析器。我们现在来简要的介绍下 Bio.SeqIO 模块--你可以在第四章找到更多内容。我们以在线查找我们的朋友 lady slipper orchids 开始。为使该介绍简单，我们使用 NCBI 网站。让我们关注下 NCBI 的核酸数据库，使用 Entrez 在线搜索任何包含有 Cypripedioideae（这是 lady slipper orchids 的亚科）的文本。(http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide) 这个简介最早写成时候，仅仅只有 94 个查询结果，我们把它保存为 FASTA 格式的纯文本文件 ls_orchid.fasta，同时保存 GenBank 格式纯文本文件 ls_orchid.gbk。如果你现在进行搜索，你会得到数百个结果！为了跟这个文档一致，以便能够查看相同的基因列表，下载以上两个文件，把它们放在 docs/examples/下即可。在 2.5 节中，我们将看下如何使用 python 进行一个像这样的搜索。

2.4.1 简单的 FASTA 分析例子

如果你使用你最喜欢的文本编辑器打开 lady slipper orchids 的 FASTA 文件，你将看到文件是像这样开始的：

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

ACCCTGATTTGTTGTTGGG

...

它包含有 94 条记录，每一个都以“>”开始，然后是在一行或多行内的序列。现在在 python 中键入：

```
from Bio import SeqIO
handle = open("ls_orchid.fasta")
for seq_record in SeqIO.parse(handle, "fasta") :
    print seq_record.id
    print repr(seq_record.seq)   # repr used to change sth into string
    print len(seq_record)
handle.close()
```

在Python中用关键字import来引入某个模块

有时候我们只需要用到模块中的某个函数，只需要引入该函数即可，此时可以通过语句 from 模块名 import 函数名1,函数名2....

打印基因ID
打印字符串
序列长度

是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print name
```

执行这段代码，会依次打印names的每一个元素：

```
Michael
Bob
Tracy
```

你会在你的屏幕上得到像这样的一些输出：

gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGA
CCGTGG...CGC', SingleLetterAlphabet())
740
...
gi|2765564|emb|Z78439.1|PBZ78439
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGT
TTACT...GCC', SingleLetterAlphabet())
592

注意 FASTA 格式没有确定的字母表，故 Bio.SeqIO 使用 SingleLetterAlphabet，而不是 DNA。

2.4.2 简单的 GenBank 分析例子

现在载入 GenBank 文件--代码基本上和上面的相同，唯一不同的是我们改变了文件名和格式字符串。

```
from Bio import SeqIO
```

```
handle = open("ls_orchid.gbk")
for seq_record in SeqIO.parse(handle, "genbank") :
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)
handle.close()
```

这样会给出：

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGA
CCGTGG...CGC', IUPACAmbiguousDNA())
740
...
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGT
TTACT...GCC', IUPACAmbiguousDNA())
592
```

这次，Bio.SeqIO 可以选择一个合理的字母表，IUPAC 含糊的 DNA（参考：http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html）。你也将发现在这个例子中，使用一个较短的字符串作为序列记录的 ID。

2.4.3 我喜欢分析-请不要停止讨论它！

Biopython 有很多的分析器，每一个都基于要分析的序列格式，有其自身的特殊使用场合。最流行的文件格式已经将分析器整合到 Bio.SeqIO, 对于稀有的和非常用的文件格式，既没有分析器，同时一个旧的分析器也没有被连接。第四章有更详细的 Bio.SeqIO 描述。请同时查询 wiki 页面(http://biopython.org/wiki/SeqIO)以更新信息，或者在邮件列表中提问。wiki 页面应该包括一个最新的文件类型支持列表和更多的序列写入文件和文件格式间转化的例子。如果你对序列比对有兴趣，可以查看第五章中的 Bio.AlignIO 模块。同时，这也有个 wiki 页面支持(http://biopython.org/wiki/AlignIO)。

下一步学习查询特殊分析器的信息，然后做些很酷的事情，那会在第九节中。如果你没有找到你需要找的信息，请考虑使用帮助文档，提交一个食谱条目。

2.5 连接生物学数据库

在生物信息学中一个惯例是，你需要从生物学数据库中提取信息。手动存取是很乏味的，特别是你有很多重复性的工作要做时。Biopython 通过使用 python 脚本使一些在线数据库可用，以试图节省你的时间和精力。目前，Biopython 的脚本包括从以下数据库中提取信息：

? ExPASy –详细信息参考第八章

? Entrez from NCBI – 查看 7.10 节.

? PubMed from NCBI – 查看 9.1 节详细使用的例子。

? SCOP

这个代码是这些模块使得编写 python 代码和这些页面进行的 CGI 脚本进行作用变得简单，这样，你可以很容易的得到处理格式的结果。在某些例子中，结果会和 Biopython 分析器紧密的结合以使其能更容易的提取信息。

2.6 下一步.....?

既然你已经了解了很多，你顺利的对 Biopython 的基本使用有了一个大概了解，下面开始使用它做些有用的工作。最好是开始读源代码，看自动产生的文档。一旦你了解了你想做什么，那个库可以这样做，你需要查看下食谱，也许已经有了做你想做的事情的代码例子了。如果你知道你想做什么，但是不知道如何去做，你可以把它们贴在 Biopython 的邮件列表上。这不仅仅帮助我们回答你的问题，同时也使我们改进文档，以帮助下一个需要做此类工作的人。开始欣赏代码吧！

第三章 序列对象

生物信息学中的主要对象毫无疑问就是生物序列，在本章中，我们将介绍 Biopython 中的技巧来处理序列--Seq 对象。在第四章的序列输入/输出部分（包括 10.1 节），我们会发现 Seq 对象也被用在联合了序列信息和注释的 SeqRecord 对象中。序列从本质上将是像 AGTACACTGGT 这样的一系列字母，这看上去很自然，因为这是生物学文件格式中所看到的序列的最常见方式。在 Seq 对象和 python 序列间有两个重要的不同点。

首先，Seq 对象包含有一些相对于 python 序列的不同的方法（例如，reverse_complement（）方法在核酸序列中使用）。

第二，Seq 对象有一个重要的属性--字母表，它被用来描述序列字符串的意思，及如何解释。例如，AGTACACTGGT 是一个 DNA 序列，还是一个包含有很多 AGCT 的蛋白质序列？

3.1 序列和字母表

字母表对象也许是使 Seq 对象不仅仅是一个序列的最重要的东西。Biopython 中现在可用的字母表在 Bio.Alphabet 模块中定义。我们将使用 IUPAC 字母表（http://www.chem.qmw.ac.uk/iupac/）来处理我们喜欢的对象： DNA， RNA 以及 Proteins。Bio.Alphabet.IUPAC 提供了蛋白质，DNA 和 RNA 的基本的定义，有一个基本的 IUPACProtein 类，同时有一个扩展的 ExtendedIUPACProtein 来提供像 Asx（asparagine or aspartic acid）， “Sec” （selenocysteine）， and “Glx” （glutamine or glutamic acid）. 对于有提供基本字母的 IUPACUnambiguousDNA 和提供每一种可能的模糊字母的 ExtendedIUPACDNA，它允许改进残基。类似的，RNA 也由两种字母表形式表示： IUPACAmbiguousRNA 或者 IUPACUnambiguousRNA.

字母表类的优点有两个。第一，这给出了 Seq 对象包含的信息类型。第二，这给出了作为类型检查的一个强制的信息。既然我们知道我们所处理的，让我们看看如何使用这个类来做些有趣的工作。你可以像这样构造一个使用默认字母表的序列。

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq（ " AGTACACTGGT " ）
>>> my_seq
```

```
Seq（ 'AGTACACTGGT '， Alphabet（））
>>> my_seq.alphabet
Alphabet（）
```

但是，当你创建自己的序列对象时，需要指定字母表--这样一个明确的 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（）
```

3.2 序列像字符串一样起作用

像通常的 python 序列一样，我们可以对 Seq 对象进行很多分析，例如，得到其长度，或者重复等。

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
IUPAC.unambiguous_dna)
for index, letter in enumerate(my_seq) :
    print index, letter
print len(letter)
```

你可以像 python 序列一样存取其元素，记住 python 从零开始计数。

```
>>> print my_seq[0] #first element
>>> print my_seq[2] #third element
>>> print my_seq[-1] #list element
```

Seq 对象也有一个.count()方法，像这样：

```
>>> len(my_seq)
32
>>> my_seq.count("G")
10
>>> float(my_seq.count("G") + my_seq.count("C")) / len(my_seq)
0.46875
```

你可以使用以上语句计算 GC 百分含量，记住 Biopython 已经把计算 GC 含量的函数内置了。查看 Bio.SeqUtils 模块。

3.2.1 序列分片

让我们来看一个更复杂点的例子，进行序列分片：

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

有两点需要注意。首先，这沿袭了传统的 python 字符串。所以序列的第一个元素索引为 0（这对计算机科学来说很平常，但对生物学来说就不然）。当你进行切片时，包含第一个元素，而不包括最后一个，（包括 4，不包括 12），和 python 中的一样。当然不是世界上的每一个人都需要这种方式。主要的目的是和 python 保持一致。第二个需要注意的是，分片是在序列数据字符串上进行的，但是产生的新的另一个 Seq 对象保留了在原始 Seq 对象中的字母表信息。同样像 python 序列一样，你可以使用起始，结束和距离。例如，我们可以得到这个 DNA 序列的第一，第二和第三位置序列。

```
>>> my_seq[0::3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
```

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

当然也可以使用负的步，反向进行：

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

3.2.2 把 Seq 对象转化为字符串

如果你真的仅仅需要一个字符串，例如打印出来，或者写入一个文件，插入一个数据库等，很容易就可以做到：

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

3.2.3 核酸序列及（反转）互补

对于核酸序列，你可以很容易得得到一个序列的互补或者反相互补序列：

```
>>> my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
>>> my_seq.complement()
Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG',
IUPACUnambiguousDNA())
>>> my_seq.reverse_complement()
Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC', IUPACUnambiguousDNA())
```

所有这些操作中，字母表属性保留。如果你碰巧想做些例如，一个蛋白质序列的反向互补的话，这是很有用的：

```
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
```

```
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq.complement()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.4/site-packages/Bio/Seq.py", line 108, in complement
    raise ValueError, "Proteins do not have complements!"
ValueError: Proteins do not have complements!
```

这里会报错！

3.2.4 连接或增加序列

自然，原则上你可以将两个序列对象加在一起--就像在 python 中连接两个字符串一样。但是，你不能连接两个字母表不一致的序列，例如一个蛋白质序列和一个 DNA 序列，这会报错:

```
>>> protein_seq + dna_seq
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/local/lib/python2.4/site-packages/Bio/Seq.py", line 42, in __add__
    raise TypeError, ("incompatable alphabets", str(self.alphabet),
TypeError: ('incompatable alphabets', 'IUPACProtein()', 'IUPACUnambiguousDNA()')
```

如果你真的想这样做，需要首先给这两个序列通用的字母表：

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq
Seq('EVRNAKACGT', Alphabet())
```

这里有一个将通用核酸序列加到明确的 IUPAC DNA 序列上的例子，结果是一个模糊的核酸序列：

```
>>> 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())
```

3.3 可变的 Seq 对象

像标准的 python 字符串一样，Seq 对象是只读的，或者用 python 术语来讲，是不可变的。除了使 Seq 对象表现的像一个字符串，同时也有一个有用的默认，因为在很多生物学应用中，你需要确认你不会改变你的序列数据：

```
>>> my_seq[5] = "G"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Seq' instance has no attribute '__setitem__'
```

但是，你可以改变它成为一个可变的序列（一个可变的 Seq 对象），对它做任何处理：

```
>>> mutable_seq = my_seq.tomutable()
>>> print mutable_seq
MutableSeq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
>>> mutable_seq[5] = "T"
>>> print mutable_seq
MutableSeq('GATCGTTGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
>>> mutable_seq.remove("T")
>>> print mutable_seq
```

```
MutableSeq('GACGTTGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> print mutable_seq
MutableSeq('CGCTAAAAGCTAGGATATATCCGGGTTGCAG',
IUPACUnambiguousDNA())
```

3.4 转录和翻译

既然序列对象有意义了，下一个事就是你可以对序列进行什么操作。Bio 目录包含两个有用的模块以转录和翻译一个序列对象。这些工具是基于序列字母表的。例如，假定我们要转录一个 DNA 序列：

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

这里包含一个 unambiguous 字母表，要想转录，我们需要：

```
>>> from Bio import Transcribe
>>> transcriber = Transcribe.unambiguous_transcriber
>>> my_rna_seq = transcriber.transcribe(my_seq)
>>> print my_rna_seq
Seq('GAUCGAUGGGCCUAUAUAGGAUCGAAAAUCGC',
IUPACUnambiguousRNA())
```

新的 RNA Seq 对象字母表被自然引出，在一次证明，Seq 对象的处理并不比处理一个简单的序列困难。你同样可以逆转录一个 RNA 序列：

```
>>> transcriber.back_transcribe(my_rna_seq)
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC',
IUPACUnambiguousDNA())
```

翻译 DNA 对象，我们有一些选择。首先，我们可以使用任意数量的翻译表 -- 依赖于我们对 DNA 序列所知。Biopython 中的翻译表是从 ftp://ftp.ncbi.nlm.nih.gov/entrez/misc/data/gc.prt 中提取的信息。所以，你有很多种选择。让我们仅仅集中在两个选择上，标准翻译表和脊椎动物线粒体 DNA 表。这些表被分别标记为数字 1 和 2。既然我们知道使用什么表，我们就进行一个基本的翻译。首先，需要得到这些表中我们使用的翻译表。因为我们仍然处理我们的 unambiguous DNA 对象，我们需要考虑拿回翻译表:

```
>>> from Bio import Translate
>>> standard_translator = Translate.unambiguous_dna_by_id[1]
>>> mito_translator = Translate.unambiguous_dna_by_id[2]
```

一旦我们得到一个适当的翻译表，就可以开始翻译一个序列：

```
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA",
IUPAC.unambiguous_dna)
>>> standard_translator.translate(my_seq)
Seq('AIVMGR*KGAR', IUPACProtein())
>>> mito_translator.translate(my_seq)
Seq('AIVMGRWKGAR', IUPACProtein())
```

注意默认的翻译将会实行到一个终止密码子。如果你知道你在翻译一个 ORF（开发阅读框），想要看到终止密码子，使用 translate_to_stop 函数很容易做到：

```
>>> standard_translator.translate_to_stop(my_seq)
Seq('AIVMGR', IUPACProtein())
```

和转录相似，也很容易将一个蛋白质序列逆向翻译为一个 DNA 序列：

```
>>> my_protein = Seq("AVMGRWKGGRAAG", IUPAC.protein)
>>> standard_translator.back_translate(my_protein)
Seq('GCTGTTATGGGTCGTTGGAAGGGTGGTCGTGCTGCTGGT',
IUPACUnambiguousDNA())
```

3.5 与字符串一起工作

对于那些不想使用序列对象，在 Bio.Seq 中也有一些接受像 python 的字符串一样的模块层的函数：

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

但是，鼓励使用默认 Seq 对象的。

第四章 序列输入/输出

在本章中，我们将更详细的讨论在第二章中简要介绍的 Bio.SeqIO 模块。这是一个相对较新的界面，在 Biopython1.43 中添加的，目的是为处理多样的序列文件格式提供一个一致的简单的界面。需要和 SeqRecord 对象一起工作，它包含一个 Seq 对象（像在第三章描述的那样），而且包括注释，像标识符和描述。我们将在这一章中介绍 SeqRecord 对象的基础，但可以在 10.1 节找到更详细的。

4.1 分析或读取序列

Bio.SeqIO.parse()将序列数据读成 SeqRecord 对象。它有两个参数：

第一个参数是要读取的 handle。handle 是一个要读的打开文件，但是可以从命令行输出，或者可以从网上下载。

第二个参数是一个小写的序列特征格式--我们不会为你猜想文件格式。到 http://biopython.org/wiki/SeqIO 查看支持的文件格式。这会返回一个给出 SeqRecord 对象的迭代器，它主要用于 for 循环中。有时你会发现自己在处理仅仅包含单一记录的文件。对于此种情况，Biopython 1.45 引入了 Bio.SeqIO.read()函数。这也使用 handle 和格式作为参数。如果有一个或仅仅一个记录，将作为一个 SeqRecord 对象返回。

4.1.1 读取序列文件

总体上来将 Bio.SeqIO.parse() 是用来读取序列文件并作为 SeqRecord 对象，往往用在 for 循环中：

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

这个例子是 2.4 节的，它会载入包含 orchid DNA 序列的 FASTA 格式文件

ls_orchid.fasta. 如果想载入一个 GenBank 格式文件，例如 ls_orchid.gbk，改下文件名和格式就可以了：

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

类似的，如果你想读取另一种文件格式的文件，同时假定 Bio.SeqIO.parse() 支持该格式，你需要做的就是将格式字符串改为合适的，例如"swiss"代表 SwissProt 文件，或者"embl"代表 EMBL 文本文件。具体文件格式列表可参看 wiki 页面 (http://biopython.org/wiki/SeqIO).

4.1.2 在序列文件中重复记录

在以上的例子中，我们往往使用 for 循环来遍历记录。你可以使用 for 循环和其他支持遍历的 python 对象（列表，元组及字符串）。Bio.SeqIO 返回的对象往往是 SeqRecord 对象的遍历。你需要轮流查看每一个记录，但是仅仅一次。附加一点是当你处理大文件时，使用 iterator 会节约你的内存。除了使用 for 循环，你也可以使用.next()方法来逐步调试条目：

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

如果你使用.next()，没有更多的结果，你将得到一个特殊的 python 对象 None 或者一个 StopIteration 错误。我们也考虑了一个特殊的情况，当你的序列文件中包含多个记录，但是你仅仅想要第一个。这种情况下，以下代码就很简明：

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

一个注意事项--像这样使用.next()方法将会忽略文件中任何添加的记录。当你的文件中仅仅有一条记录时，就像这一章中的某些在线的例子，或者包含单个染色体的 GenBank 文件，那么使用新的 Bio.SeqIO.read()函数替代。这将会检查是否有其他非期待的记录存在。

4.1.3 在一个序列文件中得到记录列表

在前一节中，我们讨论了 Bio.SeqIO.parse()给出 SeqRecord 遍历，你逐个的得到了记录。最常用的是使用两一个顺序来存取记录。Python 列表数据类型很适合这个，我们可以使用 list()把记录遍历转变为一个包含 SeqRecord 对象的列表：

```
from Bio import SeqIO
handle = open("ls_orchid.gbk")
records = list(SeqIO.parse(handle, "genbank"))
handle.close()
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.seq)
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.seq)
Giving:
Found 94 records
```

```
The last record
Z78439.1
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGT
TTACT...GCC', IUPACAmbiguousDNA())
592
The first record
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGA
CCGTGG...CGC', IUPACAmbiguousDNA())
740
```

当然，你一可以继续对 SeqRecord 对象列表使用 for 循环。使用一个 list 比一个 iterator 更灵活（例如，你可以通过 list 长度来确定记录的数目），但是会使用更多的内存，因为它把记录一次性全部包括在内存里。

4.1.4 提取数据

假定你想要从 ls_orchid.gbk 文件中提取物种列表。让我们先来看下文件的第一条记录，看下物种信息存储在那个部分。

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

会给出以下的信息：

```
ID: Z78533.1
Name: Z78533
Desription: C.irapeanum 5.8S rRNA gene and ITS1 and ITS2 DNA.
/source=Cypripedium irapeanum
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta', ..., 'Cypripedium']
/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene', 'internal transcribed spacer',
'ITS1', 'ITS2']
/references=[...]
/accessions=['Z78533']
```

/data_file_division=PLN
/date=30-NOV-2006
/organism=Cypripedium irapeanum
/gi=2765658
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGA
CCGTGG...CGC', IUPACAmbiguousDNA())

我们所需要的信息 Cypripedium irapeanum 存储在注释字典，在 source 和 organism 后，因此，我们可以这样获得：

```
print first_record.annotations["source"]
```

或者：

```
print first_record.annotations["organism"]
```

一般来说，organism 是用在科学命名上（拉丁语）,source 是一个通常的名字。在这个例子中，像平常一样，它们是一致的。现在让我们遍历所有记录，构建一个每一个 orchid 序列组成的物种的列表：

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

另一个方法是使用列表 comprehension：

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

不管哪一个方法，结果是：

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

很好，这很容易，因为 GenBank 文件是使用标准方式进行注释的。现在，

假设你想从 FASTA 文件，而不是 GenBank 文件中提取物种的列表。坏消息是你需要写一些代码从记录描述行提取你想要的数据--如果信息是在文件中的第一个位置。对于这个例子，注意，如果你使用空格分解描述行，那样的话，物种处于第 1 个位置（第 0 位置是记录标识符）。因此我们可以这样做：

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

给出：

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

更简介的使用 list comprehension 方法：

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

一般来说，从 FASTA 描述行提取信息并不是十分顺利，如果使用更好的注释文件格式，例如 GenBank 或 EMBL，这种注释信息分类将会更容易处理。

4.2 从网上分析序列

在先前的章节中，我们从一个文件 handle 来分析序列数据。我们暗示 handles 并不总是来自于文件，在本节中，将使用 handles 连接网络下载序列。注意，因为你可以下载序列数据，并且分析成 SeqRecord 对象，这并意味着总是一个好的主意。一般来说，你需要下载序列，保存以备重用的。

4.2.1 从网上分析 GenBank 记录

7.6 节详细讨论了使用 EntrezEFetch 界面，但是现在我们仅仅连接到 NCBI 使用它们的 GI 号从 GenBank 中得到些 orchid 的蛋白质。首先，我们先提取一个记录。记住，当你希望 handle 包含仅仅一个记录时，使用 Bio.SeqIO.read() 函数：

```
from Bio import Entrez
form Bio import SeqIO
handle = Entrez.efetch(db="protein", rettype="genbank", id="6273291")
seq_record = SeqIO.read(handle, "genbank")
handle.close()
print "%s with %i features" % (seq_record.id, len(seq_record.features))
```

期望输出为:

```
gi|6273291|gb|AF191665.1|AF191665 with 3 features
```

NCBI 同时会让你选择其他格式，例如 FASTA 文件。如果你不关心 GenBank 文件中的注释和特性，那会是一个很好的下载选择，因为它更小些：

```
from Bio import Entrez
form Bio import SeqIO
handle = Entrez.efetch(db="protein", rettype="fasta", id="6273291")
seq_record = SeqIO.read(handle, "fasta")
handle.close()
print "%s with %i features" % (seq_record.id, len(seq_record.features))
```

期望输出：

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

现在让我们取回更多的记录。这次，handle 包含多条记录，因此我们需要使用 Bio.SeqIO.parse() 函数：

```
from Bio import Entrez
form Bio import SeqIO
handle = Entrez.efetch(db="protein", rettype="genbank",  id="6273291,6273290,6273289")
for seq_record in SeqIO.parse(handle, "genbank") :
    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()
```

将给出以下的输出：

```
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 bradtianaa
```

查看第七章关于 Bio.Entrez 模块的更多信息，确信你读了 NCBI 使用 Entrez 的指南（7.1 节）

4.2.2 从网上分析 SwissProt 序列

现在让我们使用 handle 从 ExPASy 下载一个 SwissProt 文件，更多内容在第8章。正如上面所讲，Bio.SeqIO.read()函数包含在 1.45 版本（或更高）中。

```
from Bio import ExPASy
from Bio import SeqIO
handle = ExPASy.get_sprot_raw("O23729")
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']
```

假定你的网络连接通畅，你会得到以下返回结果：

```
O23729
CHS3_BROFI
Chalcone synthase 3 (EC 2.3.1.74) (Naringenin-chalcone synthase 3).
Seq('MAPAMEEIRQAQRAEGPAAVLAIGTSTPPNALYQADYPDYYFRITKSEHL
```

TELK...GAE', ProteinAlphabet())
Length 394
['Acyltransferase', 'Flavonoid biosynthesis', 'Transferase']

4.3 序列文件作为字典

下一件事我们需要做的是，使用 python 的字典数据类型，就像在数据库里索引和定位 orchid 文件一样。对于大的文件这是很有用的，因为你仅仅需要存取文件的某个元素，做成一个好的快速的数据库。你可以使用 SeqIO.to_dict()函数来生成一个 SeqRecord 字典（在内存里）。默认的，这将使用到每一个记录的标识符（id）作为关键词。让我们用 GenBank 文件试一下：

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

由于这个可变的 orchid_dict 成为一个 python 字典，我们可以使用我们已有的所有关键词进行查找：

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

我们可以通过关键词来存取一个单 SeqRecord 对象，就像平常一样控制该对象：

```
>>> 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('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGA
TCACAT...GGT', IUPACAmbiguousDNA())
```

所以，很容易创建一个内存里的关于 GenBank 记录的数据库。接下来，我们将使用 FASTA 格式来试一下。

4.3.1 写入序列文件

我们将讨论使用 Bio.SeqIO.parse()进行序列输入（读取文件），现在，我们将看一下 Bio.SeqIO.write()进行序列输出（写入文件）。这个函数包含有三个变量：

一些 SeqRecord 对象，一个要写入的 handle 及一个序列格式。
这里有一个例子，我们以创建一些 SeqRecord 对象开始（手动，而非从一个文件载入）

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein
rec1 =
SeqRecord(Seq("MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTF
RGPSETHLDSMVGQALFGD" \
           +"GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREV
GLTFHLLKDVPGLISK" \
           +"NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKE
EKMRATREVLSEYGNM" \
           +"SSAC", generic_protein),
        id="gi|14150838|gb|AAK54648.1|AF376133_1",
        description="chalcone synthase [Cucumis sativus]")
rec2 =
SeqRecord(Seq("YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEIL
KENPSMCEYMAPSLDARQ" \
           +"DMVVVEIPKLGKEAAVKAIKEWGQ", generic_protein),
        id="gi|13919613|gb|AAK33142.1|",
        description="chalcone synthase [Fragaria vesca subsp. bracteata]")
rec3 =
SeqRecord(Seq("MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFR
ITNSEHKVELKEKFKRMC" \
           +"EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVP
KLGKEAAQKAIKEWGQP" \
           +"KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQ
GCFAGGTVLRMAKDLAEN" \
           +"NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVII
GSDPIPEVERPLFELV" \
           +"SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVE
AFQPLGISDWNSLFW" \
```

```
            +"IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVL
FILDEMRKASAKEGLGT" \
            +"TGEGLEWGVLFGFGPGLTVETVVLHSVAT", generic_protein),
        id="gi|13925890|gb|AAK49457.1|",
        description="chalcone synthase [Nicotiana tabacum]")

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

现在我们有了一个 SeqRecord 对象列表，我们将把它们写入到一个 FASTA 格式文件中。

```
from Bio import SeqIO
handle = open("my_example.fasta", "w")  # 原文中为 faa
SeqIO.write(my_records, handle, "fasta")
handle.close()
```

在你最喜欢的文本编辑器中打开这个文件，你可以得到像下面的结果：：

```
>gi|14150838|gb|AAK54648.1|AF376133_1 chalcone synthase [Cucumis sativus]
MMYQQGCFAGGTVLRLAKDLAENNRGARVLVVCSEITAVTFRGPSETHLDSM
VGQALFGD
GAGAVIVGSDPDLSVERPLYELVWTGATLLPDSEGAIDGHLREVGLTFHLLKD
VPGLISK
NIEKSLKEAFTPLGISDWNSTFWIAHPGGPAILDQVEAKLGLKEEKMRATREV
LSEYGNM
SSAC
>gi|13919613|gb|AAK33142.1| chalcone synthase [Fragaria vesca subsp. bracteata]
YPDYYFRITNREHKAELKEKFQRMCDKSMIKKRYMYLTEEILKENPSMCEYM
APSLDARQ
DMVVVEIPKLGKEAAVKAIKEWGQ
>gi|13925890|gb|AAK49457.1| chalcone synthase [Nicotiana tabacum]
MVTVEEFRRAQCAEGPATVMAIGTATPSNCVDQSTYPDYYFRITNSEHKVELK
EKFKRMC
EKSMIKKRYMHLTEEILKENPNICAYMAPSLDARQDIVVVEVPKLGKEAAQK
AIKEWGQP
KSKITHLVFCTTSGVDMPGCDYQLTKLLGLRPSVKRFMMYQQGCFAGGTVLR
MAKDLAEN
NKGARVLVVCSEITAVTFRGPNDTHLDSLVGQALFGDGAAAVIIGSDPIPEVER
```

PLFELV
SAAQTLLPDSEGAIDGHLREVGLTFHLLKDVPGLISKNIEKSLVEAFQPLGISD
WNSLFW
IAHPGGPAILDQVELKLGLKQEKLKATRKVLSNYGNMSSACVLFILDEMRKAS
AKEGLGT
TGEGLEWGVLFGFGPGLTVETVVLHSVAT

<注>：由于编辑器的问题，格式可能不怎么对，请参考原始文件。

4.3.2 指定字典的键

使用 FASTA 文件代替，用法和上面的相似：

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

这一次，键是：

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

当我们在先前的 2.4.1 节分析 FASTA 文件时，你就需要认识这些字符串。假定你更愿意使用其他作为键--像 AC 号。这给了我们一个很好的 SeqIO.to_dict() 的选择参数 key_function, 将让你定义你所使用的记录的字典的键。首先，当给出 SeqRecord 对象时，你需要写你自己的函数来得到你想要的键。一般来说，函数细节将依赖于你处理的输入记录的分类。但是对于 orchids，我们仅仅需要使用"|"，把标识符分开，然后返回第四个条目：

```
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]
```

这样，我们可以把这个函数送到 SeqIO.to_dict()函数以构建字典：

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

最后，像预期的那样，新的字典键：

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

我希望并不是十分复杂！

4.3.3 使用 SEGUID 来索引一个字典

给出另外一个处理 SeqRecord 对象字典的例子，我们需要使用 SEGUID checksum 函数(在 Biopython 1.44 中添加).这是一个最近的 checksum，冲突会很稀少 (也就是说，两个不同的序列使用一个相同的 checksum)在 CRC64checksum 上改进。再次处理 orchids 的 GenBank 文件：

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

会给出：

```
Z78533.1 JUEoWn6DPhgZ9nAyowsgtoD9TTo
Z78532.1 MN/s0q9zDoCVEEc+k/IFwCNF2pY
...
Z78439.1 H+JfaShya/4yyAj7IbMqgNkxdxQ
```

现在，重新调用 Bio.SeqIO.to_dict()函数的 key_function 变量期待一个函数将 SeqRecord 变成一个字符串。我们不能直接使用 seguid()函数，因为它将给出一个 Seq 对象（或一个字符串）但是，我们可以使用 python 的 lambda 特性创造一个“一次性”的函数给 Bio.SeqIO.to_dict()代替：

```
from Bio import SeqIO
from Bio.SeqUtils.CheckSum import seguid
seguid_dict = SeqIO.to_dict(SeqIO.parse(open("ls_orchid.gbk"), "genbank"),lambda
rec : seguid(rec.seq))
record = seguid_dict["MN/s0q9zDoCVEEc+k/IFwCNF2pY"]
print record.id
print record.description
```

那会检索到记录 Z78532.1,文件中的第二个条目。

4.4.1 在序列文件格式间转化

在先前的例子中，我们使用了一个 SeqRecord 对象列表作为 Bio.SeqIO.parse()的输入，但是它也接受 SeqRecord iterator，就像我们在 Bio.SeqIO.parse()中所得到的一样—这使得我们很简单的进行文件转化。例如，我们将读取 ls_orchid.gbk 文件，然后输出为 FASTA 格式：

```
from Bio import SeqIO
in_handle = open("ls_orchid.gbk", "r")
out_handle = open("my_example.fasta", "w")
SeqIO.write(SeqIO.parse(in_handle, "genbank"), out_handle, "fasta")
in_handle.close()
out_handle.close()
```

事实上，你可以在一行中完成，通过关闭文件 handle。 这是一个不好的格式，但是很简洁：

```
from Bio import SeqIO
SeqIO.write(SeqIO.parse(open("ls_orchid.gbk"), "genbank"), open("my_example.faa",
 "w"), "fasta")
```

4.4.2 转化序列文件成它们的反向互补

假如你有一个核酸序列文件，你希望把它转变成包含它们反向互补序列的文件。这次我们需要做些工作来转换我们得到的文件中的 SeqRecords 成适合保存的输出文件。我们将使用 Bio.SeqIO.parse()从一个文件中载入一些核酸序列开始。然后使用 Seq 对象的内置函数.reverse_complement()方法，打印出它们的反向互补。

```
from Bio import SeqIO
in_handle = open("ls_orchid.gbk")
for record in SeqIO.parse(in_handle, "genbank") :
    print record.id
    print record.seq.reverse_complement().tostring()
in_handle.close()
```

现在，如果你想保存这些反向互补序列到一个文件中，我们需要创造 SeqRecord 对象。对于此，我想写我们自己的函数会更简洁，我们可以决定如何命名我们的新记录：

```
from Bio.SeqRecord import SeqRecord
def make_rc_record(record) :
    """Returns a new SeqRecord with the reverse complement sequence"""
    rc_rec = SeqRecord(seq = record.seq.reverse_complement(), \
                       id = "rc_" + record.id, \
                       name = "rc_" + record.name, \
                       description = "reverse complement")
    return rc_rec
```

然后，我们可以使用这个来把输入记录转变为反向互补记录以输出。如果你不介意一次将所有的记录保存在内存中，这样的话，python 的 map()函数是一个非常好的方式来解决做这个：

```
from Bio import SeqIO
in_handle = open("ls_orchid.fasta", "r")
records = map(make_rc_record, SeqIO.parse(in_handle, "fasta"))
in_handle.close()
out_handle = open("rev_comp.fasta", "w")
SeqIO.write(records, out_handle, "fasta")
out_handle.close()
```

这是一个展现 list comprehension 强大功能的好地方，你可以使用最简单的形式完成冗长的相当的任务：

```
records = [make_rc_record(rec) for rec in SeqIO.parse(in_handle, "fasta")]
```

list comprehension 有一个对其内容很好的修饰，你可以添加一个条件语句：

```
records = [make_rc_record(rec) for rec in SeqIO.parse(in_handle, "fasta") if len(rec)<700]
```

这会在内存中创建一个序列长度小于 700 碱基对的反向互补记录。但是，如果你使用 Python 2.4 或更高版本，我们可以使用一个 generator 表达式来做相同的事情。但是这不是在内存中一次性创建所有的记录：

```
records = (make_rc_record(rec) for rec in SeqIO.parse(in_handle, "fasta") if len(rec)<700)
```

如果你喜欢压缩的代码，不介意不严格的文件 handle，我们可以简化这个成一长行：

```
from Bio import SeqIO
SeqIO.write((make_rc_record(rec) for rec in \
        SeqIO.parse(open("ls_orchid.fasta", "r"), "fasta") if len(rec) < 700), \
        open("rev_comp.fasta", "w"), "fasta")
```

个人观点，我认为以上的代码片段有点太压缩了，下面的代码更易阅读：

```
from Bio import SeqIO
records = (make_rc_record(rec) for rec in \
        SeqIO.parse(open("ls_orchid.fasta", "r"), "fasta") \
        if len(rec) < 700)
SeqIO.write(records, open("rev_comp.fasta", "w"), "fasta")
```

或者，对于 2.3 版本或更老版本：

```
from Bio import SeqIO
records = [make_rc_record(rec) for rec in \
        SeqIO.parse(open("ls_orchid.fasta", "r"), "fasta") \
        if len(rec) < 700]
SeqIO.write(records, open("rev_comp.fasta", "w"), "fasta")
```

第五章 序列比对输入/输出

本章中，我们将讨论 Bio.AlignIO 模块，它和前章的 Bio.SeqIO 模块很相似，不过是处理比对对象的，而不是 SeqRecord 对象。在 Biopython1.46 中是一个新的界面，目的是提供一个简单的统一的界面，处理各种序列比对文件格式。注意 Bio.SeqIO 和 Bio.AlignIO 都可以读写序列比对文件。恰当的选择将很大程度上以来对数据如何处理。

5.1 分析或读取序列比对

我们有两个函数读取序列比对 Bio.AlignIO.read()和 Bio.AlignIO.parse() ，依照在 Bio.SeqIO 里介绍的传统，分别对应于包含一个或多个比对的文件。使用 Bio.AlignIO.parse() 将会返回一个给出比对对象的 iterator。iterator 通常用在 for 循环中。这种情况的例子包括，你需要有不同的多序列比对，包括从 PHYLIP 工具 seqboot 的重取样，或者从 EMBOSS 工具 water or needle，或者 FASTA 工具。但是，在很多情况下，你需要处理包含仅仅一条比对的文件。这种情况下，你需要使用 Bio.AlignIO.read()函数，它将返回一个单比对对象。两个函数都有两个强制的理由：

1，从数据中读取的 handle，往往是一个 open 文件。

2，小写字符串指定序列格式。在 Bio.SeqIO 中，我们不会试图为你猜测文件的格式。

还有一个可选的 seq_count，将在 5.1.3 节讨论，处理包含多于 1 个比对的模糊文件格式。

5.1.1 单比对

作为一个例子，考虑下面的 PFAM 或 Stockholm 文件格式中丰富蛋白质比对的注释：

```
# STOCKHOLM 1.0
#=GS COATB_BPIKE/30-81   AC P03620.1
#=GS COATB_BPIKE/30-81   DR PDB; 1ifl ; 1-52;
#=GS Q9T0Q8_BPIKE/1-52   AC Q9T0Q8.1
#=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 Q9T0Q9_BPFD/1-49    AC Q9T0Q9.1
#=GS Q9T0Q9_BPFD/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          AEPNAATNYATEAMDSLKTQAIDLISQTWPVVT
TVVVAGLVIRLFKKFSSKA
#=GR COATB_BPIKE/30-81     SS    -HHHHHHHHHHHHHH--HHHHHHHH--
HHHHHHHHHHHHHHHHHHHHHH----
Q9T0Q8_BPIKE/1-52          AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTT
VVVAGLVIKLFKKFVSRA
COATB_BPI22/32-83          DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTS
VAVAGLAIRLFKKFSSKA
COATB_BPM13/24-72          AEGDDP...AKAAFNSLQASATEYIGYAWAMVVV
IVGATIGIKLFKKFTSKA
#=GR COATB_BPM13/24-72     SS    ---S-
T...CHCHHHHCCCCTCCCTTCHHHHHHHHHHHHHHHHHHHHCTT--
COATB_BPZJ2/1-49           AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVI
VGATIGIKLFKKFASKA
Q9T0Q9_BPFD/1-49           AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVI
VGATIGIKLFKKFTSKA
#=GR Q9T0Q9_BPFD/1-49      SS    ------...-
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH--
COATB_BPIF1/22-73          FAADDATSQAKAAFDSLTAQATEMSGYAWALVV
LVVGATVGIKLFKKFVSRA
#=GR COATB_BPIF1/22-73     SS    XX-HHHH--HHHHHH--HHHHHHH--
HHHHHHHHHHHHHHHHHHHHHHHHH---
#=GC SS_cons               XHHHHHHHHHHHHHHHHCHHHHHHHHCHHHHHHH
HHHHHHHHHHHHHHHHHC--
#=GC seq_cons              AEssss...AptAhDSLpspAT-
hIu.sWshVsslVsAsluIKLFKKFsSKA
//
```

这是 Phage_Coat_Gp8（PF05371）的 PFAM 条目的比对 seed，从 http://pfam.sanger.ac.uk/family/alignment/download/gzipped?acc=PF05371&alnType=seed 下载并压缩。我们可以像下面这样载入这个文件“PF05371_seed.sth”（假定已被保存在当前目录中）

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

这个代码将打印出比对的摘要：

```
SingleLetterAlphabet() alignment with 7 rows and 52 columns
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRL...SKA
COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKL...SRA
Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRL...SKA
COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA
COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA
COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKL...SKA
Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKL...SRA
COATB_BPIF1/22-73
```

你会发现在上面的输出序列被截断。我们需要自己写代码通过 SeqRecord 对象作为行来进格式化：

```
from Bio import AlignIO
alignment = AlignIO.read(open("PF05371_seed.sth"), "stockholm")
print "Alignment length %i" % alignment.get_alignment_length()
for record in alignment :
    print "%s - %s" % (record.seq, record.id)
```

将给出以下的比对结果：

```
Alignment length 52
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA - 
COATB_BPIKE/30-81
```

AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA - Q9T0Q8_BPIKE/1-52
DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA - COATB_BPI22/32-83
AEGDDP---AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - COATB_BPM13/24-72
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA - COATB_BPZJ2/1-49
AEGDDP---AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA - Q9T0Q9_BPFD/1-49
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA - COATB_BPIF1/22-73

你注意到上面的原始文件中，一些序列，包括 PDB 数据库交叉引用，相关的二级结构？试试这个：

```
for record in alignment :
    if record.dbxrefs :
        print record.id, record.dbxrefs
```

输出：

```
COATB_BPIKE/30-81 ['PDB; 1ifl ; 1-52;']
COATB_BPM13/24-72 ['PDB; 2cpb ; 1-49;', 'PDB; 2cps ; 1-49;']
Q9T0Q9_BPFD/1-49 ['PDB; 1nh4 A; 1-49;']
COATB_BPIF1/22-73 ['PDB; 1ifk ; 1-50;']
```

要得到所有的序列注释，试试这个：

```
for record in alignment :
    print record
```

Sanger 提供了一个很好的 web 界面：http://pfam.sanger.ac.uk/family?acc=PF05371 ，可以让你用很多其他格式下载这个比对。这是 FASTA 格式的文件：

```
>COATB_BPIKE/30-81
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIRLFKKFSSKA
>Q9T0Q8_BPIKE/1-52
AEPNAATNYATEAMDSLKTQAIDLISQTWPVVTTVVVAGLVIKLFKKFVSRA
>COATB_BPI22/32-83
```

DGTSTATSYATEAMNSLKTQATDLIDQTWPVVTSVAVAGLAIRLFKKFSSKA
>COATB_BPM13/24-72
AEGDDP...AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPZJ2/1-49
AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFASKA
>Q9T0Q9_BPFD/1-49
AEGDDP...AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTSKA
>COATB_BPIF1/22-73
FAADDATSQAKAAFDSLTAQATEMSGYAWALVVLVVGATVGIKLFKKFVSRA

假定你下载保存成 PF05371_seed.faa，然后你可以使用大致相同的代码载入它们：

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

不同的仅仅是文件名和格式字符串。你仍会像以前那样得到一样的结果，序列和标识符是相同的。但是，就像你预期的那样，如果你检查每一个 SeqRecord 会发现既没有注释也没有交叉引用，英文在 FASTA 文件格式中没有包括。注意到，宁可不使用 Sanger 网站，你也可能自己使用 Bio.AlignIO 把原始的 Stockholm 格式文件转变为 FASTA 文件（看下面的）。使用任何支持的文件格式，你可以利用相同的方式，通过改变格式字符串载入比对。例如，“phylip”对应 PHYLIP 文件，“nexus”对应 NEXUS 文件，“emboss”对应 EMBOSS 工具的比对输出。在 wiki 页上有一个详细的列表 http://biopython.org/wiki/AlignIO)。

5.1.2 多序列比对

前一节集中于包含单比对的文件的读取。但是一般来说，文件会包含多与一个比对，读取这些文件就需要用到 Bio.AlignIO.parse()函数。假定有一个小的比对是 PHYLIP 格式的：

```
 5  6
Alpha     AACAAC
Beta      AACCCC
Gamma     ACCAAC
Delta     CCACCA
Epsilon   CCAAAC
```

如果你想使用 PHYLIP 工具来自主构建一个系统进化树，其中一步是使用 bootseq 工具创建一系列很多的重复取样比对，它将给出像这样的输出，其中为

了简洁而被简略：

```
   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     AACAAC
Delta     CCCCCA
Epsilon   CCCAAC
...
   5     6
Alpha     AAAACC
Beta      ACCCCC
Gamma     AAAACC
Delta     CCCCAA
Epsilon   CAAACC
```

如果你想使用 Bio.AlignIO 读取这些文件，你需要使用：

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

将给出以下的输出，显示仍被简略：

```
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
CCCACC Delta
CCCAAA Epsilon
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAAAC Alpha
AAACCC Beta
AACAAC Gamma
CCCCCA Delta
CCCAAC Epsilon
...
SingleLetterAlphabet() alignment with 5 rows and 6 columns
AAAACC Alpha
ACCCCC Beta
AAAACC Gamma
CCCCAA Delta
CAAACC Epsilon
```

和 Bio.SeqIO.parse()函数一样，使用 Bio.AlignIO.parse()返回一个 iterator.如果你像一次性把所有比对都保存在内存中，允许你以任何顺序存取，则需要转变 iterator 成 list：

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

5.1.3 模糊比对

很多比对文件格式可以明确存储多与一条比对，每一个比对的界限是明显

的。但是，当一个一般的序列文件格式被使用时，就没有这样的块结构。最常遇见的情况是当比对被保存在 FASTA 文件格式中。例如，考虑下面的：

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

这回事一个单比对包含有六个序列，有重复的标识符。或者，从标识符来看，可能包含两个不同比对，三个序列，都有一个相同的长度。

下一个例子：

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

这也是一个包含 6 个序列的单比对。但是这次是基于标识符，我们可以猜想是三个双比对，含有相同的长度。这个最后的例子是相似的：

```
>Alpha
ACTACGACTAGCTCAG--G
>XXX
```

ACTACCGCTAGCTCAGAAG
>Alpha
ACTACGACTAGCTCAGG
>YYY
ACTACGGCAAGCACAGG
>Alpha
--ACTACGAC--TAGCTCAGG
>ZZZ
GGACTACGACAATAGCTCAGG

在第三个例子中，因为它们的长度不同，不能被作为包含 6 个记录的单比对。但是，可以是三个双比对。无疑试图在 FASTA 文件中存储多个比对是不理想的。但是，如果你强制处理这些作为输入文件，Bio.AlignIO 可以处理大多数常见的情况--所有的比对都含有相同的记录数目。关于这个的一个例子是两两比对的集合，可以通过 EMBOSS 工具 needle and water 产生--尽管在这种情况下，Bio.AlignIO 应该可以理解使用 emboss 作为格式字符串的自然输出。为了解释这些作为一些分割比对的 FASTA 例子，我们可以使用带有可选参数 seq_count（它指定了每一个比对中的序列数）的 Bio.AlignIO.parse()函数(在这些例子中,分别为 3, 2 ,2). 例如，使用第三个例子作为输入数据：

```
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
```

输出：

```
Alignment length 19
ACTACGACTAGCTCAG--G - Alpha
ACTACCGCTAGCTCAGAAG - XXX
Alignment length 17
ACTACGACTAGCTCAGG - Alpha
ACTACGGCAAGCACAGG - YYY
Alignment length 21
--ACTACGAC--TAGCTCAGG - Alpha
GGACTACGACAATAGCTCAGG - ZZZ
```

Using Bio.AlignIO.read() or Bio.AlignIO.parse() without the seq_count argument

would give a single alignment containing all six records for the first two examples. For the third example, an exception would be raised because the lengths differ preventing them being turned into a single alignment.

If the file format itself has a block structure allowing Bio.AlignIO to determine the number of sequences in each alignment directly, then the seq_count argument is not needed. If it is supplied, and doesn't agree with the file contents, an error is raised.

Note that this optional seq_count argument assumes each alignment in the file has the same number of sequences. Hypothetically you may come across stranger situations, for example a FASTA file containing several alignments each with a different number of sequences – although I would love to hear of a real world example of this. Assuming you cannot get the data in a nicer file format, there is no straight forward way to deal with this using Bio.AlignIO. In this case, you could consider reading in the sequences themselves using Bio.SeqIO and batching them together to create the alignments as appropriate.

5.2Writing Alignments（算法描述）

我们讨论了使用 Bio.AlignIO.read()和 Bio.AlignIO.parse()分析比对输入（读取文件），现在我们将看一下 Bio.AlignIO.write() ，用在比对输出上（写入文件）。这是有三个变量的函数：一些比对对象，一个要写入的 handle，一个字符串格式。这有一个例子，我们以创建一些比对对象开始（手动，而非从一个文件载入）：

```
from Bio.Align.Generic import Alignment
from Bio.Alphabet import IUPAC, Gapped
alphabet = Gapped(IUPAC.unambiguous_dna)
align1 = Alignment(alphabet)
align1.add_sequence("Alpha", "ACTGCTAGCTAG")
align1.add_sequence("Beta",  "ACT-CTAGCTAG")
align1.add_sequence("Gamma", "ACTGCTAGDTAG")
align2 = Alignment(alphabet)
align2.add_sequence("Delta",  "GTCAGC-AG")
align2.add_sequence("Epislon","GACAGCTAG")
align2.add_sequence("Zeta",   "GTCAGCTAG")
align3 = Alignment(alphabet)
align3.add_sequence("Eta",    "ACTAGTACAGCTG")
align3.add_sequence("Theta", "ACTAGTACAGCT-")
```

```
align3.add_sequence("Iota",  "-CTACTACAGGTG")
my_alignments = [align1, align2, align3]
```

现在，我们有了比对对象的一个列表，我们将把它们写成 PHYLIP 格式文件：

```
from Bio import AlignIO
handle = open("my_example.phy", "w")
SeqIO.write(my_alignments, handle, "phylip")
handle.close()
```

当你使用你最喜欢的文本编辑器打开时，它会是这样的：

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

更平常的是载入一个存在的比对，也许在经过一些简单的处理，例如移除特定行或列后，保存它。

5.3 在序列比对文件格式间转换

就像使用 Bio.SeqIO 对序列的格式进行转化一样，使用 Bio.AlignIO 也可以对序列比对文件进行格式转换，我们使用 Bio.AlignIO.parse()载入比对，然后使用 Bio.AlignIO.write()把它们保存起来。对于这个例子，我们将载入先前使用的 PFAM/Stockholm 格式文件，保存成 Clustalw 格式文件：

```
from Bio import AlignIO
alignments = AlignIO.parse(open("PF05371_seed.sth"), "stockholm")
handle = open("PF05371_seed.aln","w")
AlignIO.write(alignments, handle, "clustal")
handle.close()
```

The Bio.AlignIO.write()函数期望被给予多序列比对对象。在上面的例子中，

我们给了它用 Bio.AlignIO.parse()返回的比对 iterator。这种情况下，我们知道文件中仅有一个比对，这样我们也可以使用 Bio.AlignIO.read()，但是注意我们需要把这个比对作为单比对 list 传输给 Bio.AlignIO.write()：

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

不管那种方式，我们需要以一样的新的 Clustalw 格式文件 PF05371_seed.aln 结尾，使用下面的内容：

```
CLUSTAL X (1.81) multiple sequence alignment

COATB_BPIKE/30-81               AEPNAATNYATEAMDSLKTQAIDLISQTWPV
VTTVVVAGLVIRLFKKFSS
Q9T0Q8_BPIKE/1-52               AEPNAATNYATEAMDSLKTQAIDLISQTWPV
VTTVVVAGLVIKLFKKFVS
COATB_BPI22/32-83               DGTSTATSYATEAMNSLKTQATDLIDQTWPV
VTSVAVAGLAIRLFKKFSS
COATB_BPM13/24-72                                   AEGDDP---
AKAAFNSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPZJ2/1-49                                    AEGDDP---
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFAS
Q9T0Q9_BPFD/1-49                                    AEGDDP---
AKAAFDSLQASATEYIGYAWAMVVVIVGATIGIKLFKKFTS
COATB_BPIF1/22-73               FAADDATSQAKAAFDSLTAQATEMSGYAWAL
VVLVVGATVGIKLFKKFVS
COATB_BPIKE/30-81                KA
Q9T0Q8_BPIKE/1-52                RA
COATB_BPI22/32-83                KA
COATB_BPM13/24-72                 KA
COATB_BPZJ2/1-49                 KA
Q9T0Q9_BPFD/1-49                 KA
COATB_BPIF1/22-73                RA
```

另外，也可以创建一个 PHYLIP 格式文件，命名为 PF05371_seed.phy：

```
from Bio import AlignIO
alignment = AlignIO.read(open("PF05371_seed.sth"), "stockholm")
handle = open("PF05371_seed.phy","w")
AlignIO.write([alignment], handle, "phylip")
handle.close()
```

这一次，输出是像这样的：

```
 7 52
COATB_BPIK  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIRLFKKFSS
Q9T0Q8_BPI  AEPNAATNYA TEAMDSLKTQ AIDLISQTWP VVTTVVVAGL VIKLFKKFVS
COATB_BPI2  DGTSTATSYA TEAMNSLKTQ ATDLIDQTWP VVTSVAVAGL AIRLFKKFSS
COATB_BPM1  AEGDDP---A KAAFNSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPZJ  AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFAS
Q9T0Q9_BPF  AEGDDP---A KAAFDSLQAS ATEYIGYAWA MVVVIVGATI GIKLFKKFTS
COATB_BPIF  FAADDATSQA KAAFDSLTAQ ATEMSGYAWA LVVLVVGATV GIKLFKKFVS
            KA
            RA
            KA
            KA
            KA
            KA
            RA
```

一个很大的不便是 PHYLIP 比对文件格式，序列标识符被强制分为 10 个一组。在这个例子中，正像你看到的，结果名字仍是唯一的--但是它们并不容易读取。在这种特殊情况下，没有一个清晰的方式来压缩标识符，但是为了变量，你可能像分配你自己的名字或数字系统。下面的一点代码在保存输出前，控制了记录标识符：

```
from Bio import AlignIO
```

```
alignment = AlignIO.read(open("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
handle = open("PF05371_seed.phy","w")
AlignIO.write([alignment], handle, "phylip")
handle.close()
```

这个代码使用了 python 字典来记录一个简单的 mapping，从新序列系统到原始标识符：

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

Here is the new PHYLIP format output:

```
 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
GIKLFKKFTS
seq4        AEGDDP---A  KAAFDSLQAS  ATEYIGYAWA  MVVVIVGATI
GIKLFKKFAS
seq5        AEGDDP---A  KAAFDSLQAS  ATEYIGYAWA  MVVVIVGATI
GIKLFKKFTS
seq6        FAADDATSQA  KAAFDSLTAQ  ATEMSGYAWA  LVVLVVGATV
GIKLFKKFVS
        KA
        RA
        KA
        KA
        KA
        KA
```

RA

一般来说，因为标识符的限制，使用 PHYLIP 文件格式不是你的第一个选择。另一方面，使用 PFAM/Stockholm 格式也使你记录很多附加的注释。

第六章 BLAST

嘿，每一个人都喜欢 BLAST，是吗？我的意思是，如何更容易的比较你自己的序列和世界上其他已知的序列？但是，这一节不是关于 BLAST 是什么，因为我们已经知道了。它是关于 BLAST 的问题的--它真的很困难，处理大量大规模产生的数据，自动进行 BLAST。幸运的是，Biopython 对此了解很多，所以它们开发了一系列工具来处理 BLAST，以使事情变得更容易。这一节详细介绍如何使用这些工具，并利用它们做些有用的事情。处理 BLAST 可以分为两步，都可以在 Biopython 中进行。首先，对你的查询序列运行 BLAST，得到一些输出。其次，在 python 中分析 BLAST 输出以应对未来分析。我们将以本地运行 BLAST 命令行开始讨论，然后讨论通过网络的 BLAST。

6.1 本地运行 BLAST

本地运行 BLAST（和网络运行相反，查看 6.2 节）有两个优点：

1，本地运行 BLAST 可能比通过网络运行 BLAST 要快。

2，本地运行 BLAST 使你可以构建自己的数据库以进行搜索

处理私有的或未发表的序列数据可能是本地运行 BLAST 的另一个原因。你可能不允许重新分配序列，因此提交到 NCBI 上作为 BLAST 检索不是一个好的选择。Biopython 提供了很多好的代码使你能够从你的脚本来访问本地 BLAST 执行，提供有很全的执行命令行选项。你可以获得多平台的本地 BLAST 预编译文件在 ftp://ftp.ncbi.nlm.nih.gov/blast/executables/,或者你自己在 NCBI 工具包中编译它 (ftp://ftp.ncbi.nlm.nih.gov/toolbox/)。处理本地 BLAST 的代码在 Bio.Blast.NCBIStandalone 中，特别是 blastall, blastpgp 和 rpsblast,它们的名字与 BLAST 执行像一致。让我们使用这些函数做一个本地数据库的 BLAST，然后返回结果。首先，我们需要设置我们需要运行 BLAST 时的路径。我们需要知道的是检索数据库的路径（使用 formatdb 准备好，参考 ftp://ftp.ncbi.nlm.nih.gov/blast/documents/formatdb.html)，我们需要查找的文件的路径，blastall 执行的路径。在 Linux 或 Mac OS X，你的路径会像这样：

```
>>> my_blast_db = "/home/mdehoon/Data/Genomes/Databases/bsubtilis"
# I used formatdb to create a BLAST database named bsubtilis
# (for Bacillus subtilis) consisting of the following three files:
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nhr
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nin
```

```
# /home/mdehoon/Data/Genomes/Databases/bsubtilis.nsq
>>> my_blast_file = "m_cold.fasta"
# A FASTA file with the sequence I want to BLAST
>>> my_blast_exe = "/usr/local/blast/bin/blastall"
# The name of my BLAST executable
```

在 Windows 上可能是这样：

```
>>> my_blast_db = r"C:\Blast\Data\bsubtilis"
# Assuming you used formatdb to create a BLAST database named bsubtilis
# (for Bacillus subtilis) consisting of the following three files:
# C:\Blast\Data\bsubtilis\bsubtilis.nhr
# C:\Blast\Data\bsubtilis\bsubtilis.nin
# C:\Blast\Data\bsubtilis\bsubtilis.nsq
>>> my_blast_file = "m_cold.fasta"
>>> my_blast_exe =r"C:\Blast\bin\blastall.exe"
```

这个例子中使用的 FASTA 文件在这里和网络都可用。既然我们已经将所有都设定了，准备运行 BLAST，收集结果。我们可以使用两行来进行：

```
>>> from Bio.Blast import NCBIStandalone
>>> result_handle, error_handle = NCBIStandalone.blastall(my_blast_exe, "blastn",my_blast_db,my_blast_file)
```

注意对于本地 BLAST，Biopython 界面返回两个值。第一个是输出的准备保存或者进行分析的 BLAST handle。第二个是可能的由 blast 命令产生的错误输出。参考 12.1 节获得更多关于 handles 的信息。错误信息很难处理，因为如果你试图做 error_handle.read()，没有错误信息返回，然后 read()将锁定不返回，锁定你的脚本。我认为最好的处理错误的方式就是如果你得不到 result_handle 结果分析的话打印出它，或者不管它。这个命令将产生以 XML 格式的 BLAST 输出，这是 XML 分析器期望的格式，在 6.4 节中有描述。对于纯文本输出，使用 align_view='0' 关键词。分析纯文本输出而不是 XML 输出，请参考 6.6 节。但是，不推荐分析纯文本输出，因为 BLAST 纯文本输出经常改变，会打破我们的分析器。如果你对分析输出结果前把它们存储到文件中更感兴趣，参考 6.3 节。如何分析 BLAST 结果，转到 6.4 节。

6.2 在网络上运行 BLAST

自动 BLAST 的第一步是保证从 python 脚本中所有事都容易取得。故 Biopython 包含有让你直接使用你的 python 脚本运行网络版的 BLAST

(http://www.ncbi.nlm.nih.gov/BLAST/) 。这是很好的，特别是当其他的 BLAST 让人感觉不舒服时，特别是对于全 BLAST 检索，分割的结果页等。处理网络版 BLAST 的代码在 Bio.Blast.NCBIWWW 模块和 qblast 函数中。我们希望使用 FASTA 格式文件对数据库进行 blast。首先，我们需要得到 FAST 文件中的信息。最简单的方法是使用 Bio.SeqIO 模块（参考第四章）。在这个例子中，我们将使用 Bio.SeqIO.read 函数将一个包含单条目的 FASTA 文件转变成一个 SeqRecord 对象：

```
>>> from Bio import SeqIO
>>> record = SeqIO.read(open("m_cold.fasta"), format="fasta")
```

现在我们可以从 SeqRecord 对象中，将序列作为一个普通字符串，然后对它运行 BLAST。运行 BLAST 的简单代码如下：（对一个非冗余数据库运行 blastn）

```
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nr", record.seq.tostring())
```

对于 NCBIWWW.qblast 函数的前三个变量是非选择的：

第一个变量是要进行检索的 blast 程序，作为一个小写的字符串。对于这些的描述可以参考 http://www.ncbi.nlm.nih.gov/BLAST/blast_program.html. 目前 qblast 仅仅支持 blastn, blastp, blastx, tblast 及 tblastx.

第二个参数标识要进行搜索的数据库。在 http://www.ncbi.nlm.nih.gov/BLAST/blast_databases.html 上有说明。（即使用哪个数据库）

第三个参数是包含要检索的序列的字符串。这可以是一个序列，一个 FASTA 格式的序列，或者像 GI 一样的标识符。

qblast 函数也有其他选项的变量， 基本类似于在 BLAST 网页上得到的不同参数。我们在这里加亮它们中的一些：

qblast 函数可以以多种格式返回 BLAST 结果，格式类型关键字包括："HTML", "Text", "ASN.1", 或者 "XML"。默认是 "XML", 这是分析器期待的格式，在 6.4 节有描述。

变量期望设置设置 e-value 阈值。对于更多可选的 BLAST 变量，请查看 NCBI 的文档，或者在 biopython 中调用：

```
>>> from Bio.Blast import NCBIWWW
>>> help(NCBIWWW.qblast)
```

在你设定搜索选项后，就准备好了 BLAST. Biopython takes care of worrying about when the results are available, and will pause until it can get the results and return them.

6.3 保存 BLAST 输出结果

在分析结果之前，把它们存储到一个文件里是很有用的，这样你可以过后使用它们，而不是对所有的事重新进行 blast。当调试我的代码从 BLAST 文件中提取信息时，我感觉这很有意思，但是，对于备份你所做的也是很有用的。如果你不想保存 BLAST 输出，直接跳到 6.4 节。如果需要，请继续阅读。我们需要优点谨慎，因为我们只可以使用 result_handle.read()读取 blast 输出一次--再次调用 result_handle.read()将返回一个空字符串。我们首先使用 read()把来自 handle 的所有信息存成一个字符串：

```
>>> blast_results = result_handle.read()
```

接下来，我们保存这个字符串到一个文件：

```
>>> save_file = open("my_blast.xml", "w")
>>> save_file.write(blast_results)
>>> save_file.close()
```

做了这以后，结果在 my_blast.xml 中，可用的包含 blast 结果的 blast_result 在一个字符串结构中。但是，blast 分析器的分析函数带有一个文件句柄式的对象，不是一个普通字符串（plain string）。为得到一个 handle，需要做两件事：使用 python 标准库的 cStringIO 模块。下面的代码将把一个普通字符串变成一个 handle，我们可以直接提供给 blast 分析器：

```
>>> import cStringIO
>>> result_handle = cStringIO.StringIO(blast_results)
```

打开保存的文件以读取。

```
>>> result_handle = open("my_blast.xml")
```

既然我们已经得到了 blast 结果，我们需要对其进行处理，这将把我们带到分析部分。

6.4 分析 BLAST 输出

如上所述，blast 可以产生很多输出格式，像 XML, HTML, 及纯文本。最初，Biopython 有一个对 BLAST 纯文本和 HTML 输出的分析器，这些是 BLAST 支持的唯一输出格式。不幸的是，这些格式的 BLAST 输出总是在改变，每次都会破坏 biopython 分析器。要跟上 blast 的改变是一个无希望的尝试，特别是对于运行不同 BLAST 版本的用户，我们建议分析输出为 XML 格式，它可以使用最新

版本的 BLAST 产生。XML 输出不仅比普通文本和 HTML 输出更稳定，而且更容易自动分析，使得 biopython 更加稳定。尽管不赞成，以普通文本或 HTML 格式的输出在 biopython 中仍可使用。使用它们是有风险的，它们可能工作，也可能不工作，依赖于你使用的 blast 版本。你可以用很多方式获得 XML 格式 blast 输出。对于分析器，它不介意输出是如何产生的，只要它是 XML 格式就行。

? 你可以使用 Biopython 在本地运行 BLAST，见 6.1 节。

? 你可以使用 Biopython 通过网络运行 BLAST，见 6.2 节。

? 你可以通过你的浏览器在 NCBI 网站上自己运行 BLAST，然后保存结果。你需要选择 XML 格式作为返回的结果，保存你最终得到的 BLAST 页面（包括你感兴趣的结果）到一个文件。

? 你也可以不使用 Biopython 在本地运行 BLAST，把输出结果保存到一个文件。你需要选择 XML 格式作为返回的结果。

重要的一点是你不需要使用 Biopython 来取数据以分析。

以这些方式中的一个来做事情，你需要得到一个结果的 handle。在 python 中，一个 handle 仅仅是一个描述输入成任何信息来源的一个好方法，这样可以使用 read()和 readline()函数来检索信息。这是 blast 分析器用来输入的类型。如果你像上面的脚本那样，通过一个脚本来和 BLAST 进行交互，则你已经有了 result_handle，blast 结果的 handle。例如：

```
>>> from Bio import SeqIO
>>> record = SeqIO.read(open("m_cold.fasta"), format="fasta")
>>> from Bio.Blast import NCBIWWW
>>> result_handle = NCBIWWW.qblast("blastn", "nr", record.seq.tostring())
```

如果你以另外的方式运行 blast，你会在 my_blast.xml 中得到 blast 输出，你需要做的就是打开文件，然后读取：

```
>>> result_handle = open("my_blast.xml")
```

既然我们已经得到一个 handle，我们准备分析输出。分析的代码是很短小的：

```
>>> from Bio.Blast import NCBIXML
>>> blast_records = NCBIXML.parse(result_handle)
```

为了解 NCBIXML.parse 的返回，你需要牢记两点：

blast 输出可能包含不止一个 blast 搜索的输出。这将会是，例如你使用包含不止一条序列的 FASTA 文件运行本地 blast。对于每一个序列，blast 分析器将返回一个 blast 记录。

blast 输出可能会因此很大。

为了能够解决这些情况，NCBIXML.parse 返回一个 iterator（就像

Bio.SeqIO.parse).用直率的英语，一个 iterator 允许你单步调试 blast 输出，为每一个 blast 搜索，逐个检索 blast 记录：

```
>>> 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，你可以使用一个 for 循环：

```
>>> for blast_record in blast_records:
...     # Do something with blast_record
```

注意你可以单步调用 blast 记录仅一次。通常，对于每一个 blast 记录，你应该保存你感兴趣的信息。如果你保存所有返回的 blast 记录，你可以把 iterator 转变为一个 list：

```
>>> blast_records = list(blast_records)
```

现在你可以像平常对 list 使用 index 一样检索每一个 blast 记录。如果你的 blast 文件很大，你试图保存它们到一个 list 中时会出现问题。通常，你将逐次运行 blast 检索。这样，所有你需要做的就是从 blast_records 中提取第一个 blast 记录：

```
>>> blast_record = blast_records.next()
```

我猜到现在你想知道 BLAST 记录中是什么。

6.5 BLAST 记录类

一个 BLAST 记录包含你想从其结果中得到的几乎所有的事情。现在我们将给出一个例子，看下如何从 BLAST 报告中得到信息。如果你想得到一些这里没有描述的特殊信息，详细查看记录类，看一看代码或者自动产生的文档--文档中含有存储每一条信息中所包含有用信息。继续我们的例子，让我们打印出击中的 BLAST 报告项中，大于某一阈值的一些摘要信息。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] + '...'
```

将得到以下的信息：

```
****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
tacttgttgatattggatcgaacaaactggagaaccaacatgctcacgtcacttttagtcccttacatattcctc...
||||||||| | |||||||||| || ||||  || || |||||||| |||||| |  | ||||||| ||| ||...
tacttgttggtgttggatcgaaccaattggaagacgaatatgctcacatcacttctcattccttacatcttcttc...
```

基本上，一旦你分析后，你几乎可以对 BLAST 报告中的信息做任何事情。当然，这将依赖于你使用它来做什么，但是希望这会帮助你开始做你需要的事情。从 BLAST 报告中提取信息的一个重要考虑是，存储信息的对象的类型。在 biopython 中，分析器返回 Record 对象，blast 或位点专一性 psiblast，依赖于你分析什么。这些对象在 Bio.Blast.Record 中被定义，而且很完整。我 blast 和 psiblast 记录类的 UML 类图做了尝试。如果你对 UML 很熟悉，并且在错误/改进中有帮助，请让我知道。blast 类图在图 6.5 显示。PSIBlast 记录对象是相似的，但是支持在 PSIBlast 的重复步骤中使用的整个过程。PSIBlast 的类图在图 6.5 显示。

6.6 Deprecated BLAST parsers

早期版本的 biopython 使用普通的文本或 HTML 格式分析 BLAST 输出。多年来，我们发现维持这些分析器呈工作顺序是很困难的。基本上，任何新版本的 BLAST 的一个小的改变，就会使普通文本和 HTML 分析器失效。因此，我们建议分析 blast 输出为 XML 格式，像在 6.4 节中描述的那样。但是，普通文本和 HTML

分析器在 biopython 中仍然可以使用，有些风险罢了。基于你使用的 blast 版本，它们可能会也可能不会工作。

6.6.1 分析普通文本 BLAST 输出

普通文本 BLAST 分析器在 Bio.Blast.NCBIStandalone 中。像对待 XML 分析器一样，我们需要使用 handle 对象以传递给分析器。handle 必须执行 readline() 方法，正确的执行这。常用的获得 handle 的方法是使用提供的 blastall 或者 blastpgp 函数来运行本地 blast，或者通过命令行运行本地 blast，然后像下面这样做些事情：

```
>>> result_handle = open("my_file_of_blast_output.txt")
```

好了，现在我们得到一个 handle（记作 result_handle)，我们准备分析它。可以使用下面的代码来进行：

```
>>> from Bio.Blast import NCBIStandalone
>>> blast_parser = NCBIStandalone.BlastParser()
>>> blast_record = blast_parser.parse(result_handle)
```

这将会分析 blast 报告成一个 BLAST 记录类（blast 或 psiblast 记录，基于你分析的内容)，这样你就可以从它来提取信息。就本件事，我们仅仅使用打印所有的大于阈值的比对的快速摘要。

```
>>> E_VALUE_THRESH = 0.04
>>> for alignment in b_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] + '...'
```

如果你也读了 6.4 节中的分析 XML 格式的 BLAST，你会发现代码和那一节的一致。一旦你分析一些东西成一个记录类，你可以处理它而不依赖于你分析的原始 blast 信息的格式。很漂亮！当然，分析一个记录是很好，但是我有包含很多记录的 BLAST 文件--我如何全部分析呢？好吧，不用怕，下一节会有答案的。

6.6.2 分析一个包含全部 BLAST 运行的文件

当然，本地 blast 很酷，因为你可以对一个数据库运行一个整体的序列的 blast，获得一个关于它是所有的很好的报告。因此，biopython 一定有使分析极大文件变得容易，而没有内存问题的能力。我们可以使用 blast iterator 来做这个。要设置一个 iterator，我们首先设置一个分析器，来分析以 blast 记录对象形式给出的 blast 报告：

```
>>> from Bio.Blast import NCBIStandalone
>>> blast_parser = NCBIStandalone.BlastParser()
```

然后，我们假定有一个 handle 来处理大量 blast 记录，记作 result_handle。获得一个 handle 在 blast 分析节中详细描述过。既然我们已经获得一个分析器和一个 handle，准备使用下面的命令设置 iterator：

```
>>> blast_iterator = NCBIStandalone.Iterator(blast_handle, blast_parser)
```

第二个选项--分析器--是可选的。如果我们不提供一个分析器，iterator 将仅会一次返回一个原始 blast 报告。既然我们已经有了一个 iterator，我们开始使用 next()来检索 blast 记录：

```
>>> blast_record = blast_iterator.next()
```

每次调用一个 next 将返回我们处理的一个新的记录。现在我们可以重复我们的记录，产生一个好的，小点的 blast 报告：

```
>>> for b_record in b_iterator :
...     E_VALUE_THRESH = 0.04
...     for alignment in b_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
```

iterator 允许处理巨大的 blast 记录，而不会有任何内存问题,因为每次只读取一个。我就使用这个分析了十分巨大的文件。

6.6.3 在一个巨大文件中找到一个错误的记录

一个对我来说经常发生的问题是，我经常分析巨大的 blast 文件，分析器会引发一个 ValueError。这是一个严重的问题，因为你不能获知 ValueError 是由于分析器问题，还是由于 BLAST 问题导致的。更糟的是，你不知道分析在哪里出错的，所以你不能忽略错误。因为这可能会忽视一个重要的数据点。我们曾经创建了一个小脚本以绕开这个问题，但是 Bio.Blast 模块现在包含一个 BlastErrorParser，使得这更加容易。BlastErrorParser 和常规的 BlastParser 十分相似，不过通过捕获由分析器产生的 ValueError 添加了一个额外的工作层，以试图诊断错误。让我们来看看使用这个分析器--首先我们定义需要分析的文件及要写入的错误报告文件。

```
>>> 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")
```

现在我们需要获得一个 BlastErrorParser:

```
>>> from Bio.Blast import NCBIStandalone
>>> error_handle = open(error_file, "w")
>>> blast_error_parser = NCBIStandalone.BlastErrorParser(error_handle)
```

注意分析器对于 handle 有一个可选的变量。如果一个 handle 被 pass，那么这个分析器将写入由这个 handle 所产生的 ValueError 的任何 blast 记录。否则，这些记录将不会被记录。现在我们可以像常规 blast 分析器一样使用 BlastErrorParser。特别的，我们会创建一个 iterator，一次一个遍历我们的 blast 记录，使用错误分析器来分析它们：

```
>>> result_handle = open(blast_file)
>>> iterator = NCBIStandalone.Iterator(result_handle, blast_error_parser)
```

我们可以一次一个的读取这些记录，但是现在我们可以捕获和处理由于 blast 问题产生的错误（而不是分析器的问题）：

```
>>> try:
...     next_record = iterator.next()
... except NCBIStandalone.LowQualityBlastError, info:
...     print "LowQualityBlastError detected in id %s" % info[1]
```

.netx()方法通常是直接通过 for 循环调用的。目前，BlastErrorParser 可产生如下的错误：

ValueError – 这和常规 BlastParser 产生的错误一样，是因为分析器不能分析一个特定的文件。这通常是由于分析器中的一个 bug，或者是由于你使用的 BLAST 版本和分析器能够处理的版本之间的某种不一致造成的。

LowQualityBlastError – 当 blast 一个质量很差的序列（例如，仅仅一小段和一个核酸），好像 blast 以掩出整个序列，以没有什么可分析而结束。这种情况下，它将会产生一个导致分析器产生 ValueError 的删减的报告。LowQualityBlastError 以这些情况而被报告。这个错误返回一个如下面一样的信息条目：

o item[0] – 错误信息

o item[1] – 导致错误的输入记录的 ID。如果你想记录所有导致问题的记录的话，这是很有用的。

正如提到的，对于产生的每个错误，BlastErrorParser 将会写令人讨厌的记录到特定的 error_handle。你可以继续前进，当你认为合适时处理它们。你不但要能用一个 blast 记录来调试分析器，还要在你的 blast 运行中发现问题。不管那种方式，都会是一个有用的经验。希望 BlastErrorParser 可以使调试和处理大的 blast 文件更容易。

6.7 处理 PSIBlast

我们需要写些脚本来直接处理 PSIBlast 更容易（也就是说，从一个比对中使用合适的格式输出比对）。我需要更多的查看 PSIBlast 以赶得上做这些...

第七章 访问 NCBI 的 Entrez 数据库

Entrez (http://www.ncbi.nlm.nih.gov/Entrez) 是一个给用户提供使用 NCBI 的数据库，例如 PubMed, GenBank, GEO, 等的数据检索系统。你可以使用浏览器自己输入检索项进行检索 Entrez，或者使用 Biopython 的 Bio.Entrez 模块程序化的检索 Entrez。后者允许你从一个 python 脚本，例如进行检索 PubMed 或者下载 GenBank 记录。Bio.Entrez 模块使用 Entrez 程序设计应用，由 8 个工具组成将会在 http://www.ncbi.nlm.nih.gov/entrez/utils/ 详述。这些工具的每一个都和 Bio.Entrez 模块中的 python 函数相对应，正如在下面章节中描述的那样。这个模块确定在检索中使用了正确的 URL 地址，像 NCBI 评论的那样，每 3 秒钟，不少于有一个检索？？？Entrez 程序设计应用的返回结果是 XML 格式的。要分析这种输出，你有多种选择：

1，使用 Bio.Entrez 的分析器分析 XML 输出成一个 python 对象；
2，使用 python 标准库的 DOM（文档对象模型）分析器
3，使用 python 标准库的 SAX（分析 XML 的简单的 API）分析器
4，将 XML 输出读成原始文本，通过字符串搜索和处理进行分析。

对于 DOM 和 SAX 分析器，查看 Python 文档。Bio.Entrez 的分析器在下面讨论。

对于序列数据库，Entrez 程序设计应用可以以其他格式产生输出（例如 fasta 或 genbank 格式）。这个可以使用 Bio.SeqIO 分析成一个 SeqRecord 对象。

7.1Entrez 指南

在使用 biopython 检索 NCBI 在线资源之前（通过 Bio.Entrez 或其他模块），请阅读下 NCBI 的 Entrez 用户需知。如果 NCBI 发现你滥用它们的系统，它们可以并且将会限制你的访问。

解释意义：

对于大于 100 个任意连续的查询请求，在周末或者非 USA 高峰时间进行。这是你需要遵守的。

使用 http://eutils.ncbi.nlm.nih.gov/ 地址，而非标准的 NCBI 网络地址。biopython 使用这个地址。

每 3 秒内不要多于一个查询请求。这在 biopython 中是自动强制的。

使用可选的 email 参数，这样当出现问题时，NCBI 可以联系你。在下面的例子中，我们使用"A.N.Other@example.com" 以说明此意。example.com 是一个

为特定文档保留的地址。(RFC 2606). 不要使用任意邮箱--还不如不给。
如果你在一些较大的软件包中使用 biopython，使用工具参数以指定这个。对 biopython 来说，工具参数将是默认的。

对于大的查询请求，NCBI 也建议使用它们的历史特色。(the WebEnv session cookie string). 这稍微优点复杂。

综上，请合理使用你的权限。如果你打算下载很多数据，考虑其他选择。例如，如果你想轻松查询所有人类基因，考虑作为一个 GenBank 文件，通过 FTP 下载每一个染色体。把它们输入到你自己的 BioSQL 数据库。（参考 9.5 节).

7.2EInfo: 获取 Entrez 数据库的信息

EInfo 提供了对每一个 NCBI 数据库的索引栏，最后更新及可用链接。另外，你可以使用 EInfo，通过 Entrez 设施，获得所有数据库名字的一个列表：

```
>>> from Bio import Entrez
>>> handle = Entrez.einfo(email="A.N.Other@example.com")
>>> result = handle.read()
```

这个可用的结果现在包含一个 XML 格式的数据库的列表：

```
>>> 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>
```

因为这是一个很简单的 XML 文件，我们可以简单的使用字符串搜索来提取它所包含的信息。使用 Bio.Entrez 的分析器，我们可以直接分析这个 XML 文件成 python 对象：

```
>>> from Bio import Entrez
>>> handle = Entrez.einfo(email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
```

现在记录是有一个键的字典：

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

键对应的值是在 XML 文件中显示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']
```

对于这些数据库的每一个，我们可以使用 EInfo 来获得更多信息：

```
>>> handle = Entrez.einfo(db="pubmed", email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["DbInfo"]["Description"]
'PubMed bibliographic record'
>>> record["DbInfo"]["Count"]
'17989604'
>>> record["DbInfo"]["LastUpdate"]
'2008/05/24 06:45'
```

用 record["DbInfo"].keys()以在这个记录中查找其他信息。

7.3 ESearch: 搜索 Entrez 数据库

我们使用 Bio.Entrez.esearch()来搜索这些数据库的任一个。例如，让我们搜索在 PubMed 中以发表的和 Biopython 相关的内容：

```
>>> from Bio import Entrez
>>> handle = Entrez.esearch(db="pubmed", term="biopython",
email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["IdList"]
['16403221', '16377612', '14871861', '14630660', '12230038']
```

在这个输出中，你看到 5 个 PubMedID 号（16403221, 16377612, 14871861, 14630660, 12230038），它们可以通过 EFetch 来检索（查看 7.6 节）。你也可以使用 ESearch 来搜索 GenBank。这里我们将快速查找 Opuntia 的 rpl16 基因：

```
>>> handle = Entrez.esearch(db="nucleotide",term="Opuntia and
```

```
rpl16",email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["Count"]
'9'
>>> record["IdList"]
['57240072', '57240071', '6273287', '6273291', '6273290','6273289', '6273286',
'6273285', '6273284']
```

每一个 IDs (57240072, 57240071, 6273287...)都是 GenBank 的标识符。查看 7.6 节以了解如何下载这些 GenBank 记录。最后一个例子，让我们得到一系列计算机期刊的题目列表：

```
>>> handle = Entrez.esearch(db="journals",
term="computational",email="A.N.Other@example.com")
>>> 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']
```

此外，我们可以使用 EFetch 来获得每一个期刊 ID 的更详细信息。ESearch 有很多有用的选项--查看 ESearch 帮助以获得更多信息。

7.4EPost

EPost 张贴了一个在随后的搜索策略中使用的 UI 列表。查看 EPost 帮助页面以获得更多信息。在 biopython 中通过 Bio.Entrez.epost()可用。

7.5 ESummary: 从主要的 ID 来检索摘要

ESummary 从主要的 ID 来检索文档摘要（更多信息，查看 ESummary 帮助页面）在 Biopython 中，ESummary 作为 Bio.Entrez.esummary()而可用。使用上面的搜索结果，我们可以找到更多关于 ID 30367 的期刊。

```
>>> from Bio import Entrez
>>> handle = Entrez.esummary(db="journals", id="30367",
email="A.N.Other@example.com")
```

```
>>> record = Entrez.read(handle)
>>> record[0]["Id"]
'30367'
>>> record[0]["Title"]
'Computational biology and chemistry'
>>> record[0]["Publisher"]
'Pergamon,'
```

7.6 EFetch: 从 Entrez 下载所有记录

EFetch 用来从 Entrez 下载你想要的所有记录。对于上面的 Opuntia 例子，我们可以使用 Bio.Entrez.efetch 下载 GenBank 条目 57240072：

```
>>> handle = Entrez.efetch(db="nucleotide", id="57240072", rettype="genbank",email="A.N.Other@example.com")
>>> print handle.read()
LOCUS       AY851612               892 bp    DNA     linear   PLN 10-APR-2007
DEFINITION  Opuntia subulata rpl16 gene, intron; chloroplast.
ACCESSION   AY851612
VERSION     AY851612.1  GI:57240072
KEYWORDS    .
SOURCE      chloroplast Austrocylindropuntia subulata
  ORGANISM  Austrocylindropuntia subulata
            Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
            Spermatophyta; Magnoliophyta; eudicotyledons; core eudicotyledons;
            Caryophyllales; Cactaceae; Opuntioideae; Austrocylindropuntia.
REFERENCE   1  (bases 1 to 892)
  AUTHORS   Butterworth,C.A. and Wallace,R.S.
  TITLE     Molecular Phylogenetics of the Leafy Cactus Genus Pereskia
            (Cactaceae)
  JOURNAL   Syst. Bot. 30 (4), 800-808 (2005)
REFERENCE   2  (bases 1 to 892)
  AUTHORS   Butterworth,C.A. and Wallace,R.S.
  TITLE     Direct Submission
  JOURNAL   Submitted (10-DEC-2004) Desert Botanical Garden, 1201 North
Galvin
```

```
        Parkway, Phoenix, AZ 85008, USA
FEATURES             Location/Qualifiers
     source          1..892
                     /organism="Austrocylindropuntia subulata"
                     /organelle="plastid:chloroplast"
                     /mol_type="genomic DNA"
                     /db_xref="taxon:106982"
     gene            <1..>892
                     /gene="rpl16"
     intron          <1..>892
                     /gene="rpl16"
ORIGIN
        1 cattaaagaa gggggatgcg gataaatgga aaggcgaaag aaagaaaaaa atgaatctaa
       61 atgatatacg attccactat gtaaggtctt tgaatcatat cataaaagac aatgtaataa
      121 agcatgaata cagattcaca cataattatc tgatatgaat ctattcatag aaaaaagaaa
      181 aaagtaagag cctccggcca ataaagacta agagggttgg ctcaagaaca aagttcatta
      241 agagctccat tgtagaattc agacctaatc attaatcaag aagcgatggg aacgatgtaa
      301 tccatgaata cagaagattc aattgaaaaa gatcctaatg atcattggga aggatggcgg
      361 aacgaaccag agaccaattc atctattctg aaaagtgata aactaatcct ataaaactaa
      421 aatagatatt gaaagagtaa atattcgccc gcgaaaattc ctttttatt aaattgctca
      481 tattttattt tagcaatgca atctaataaa atatatctat acaaaaaaat atagacaaac
      541 tatatatata taatatattt caaatttcct tatataccca aatataaaaa tatctaataa
      601 attagatgaa tatcaaagaa tctattgatt tagtgtatta ttaaatgtat atcttaattc
      661 aatattatta ttctattcat ttttattcat tttcaaattt ataatatatt aatctatata
      721 ttaatttata attctattct aattcgaatt caatttttaa atattcatat tcaattaaaa
      781 ttgaaatttt ttcattcgcg aggagccgga tgagaagaaa ctctcatgtc cggttctgta
      841 gtagagatgg aattaagaaa aaaccatcaa ctataacccc aagagaacca ga
//
```

参数 rettype="genbank"使得我们可以以 GenBank 格式下载这个条目。或者，你可以使用 rettype="fasta"得到一个 FASTA 格式的，参看 EFetch 帮助页面的了解其他选项。可用的格式依赖于你从那个数据库中下载。如果你使用 Bio.SeqIO 可接受的某种格式来下载条目，你可以直接把它分析成 SeqRecord:

```
>>> from Bio import Entrez, SeqIO
>>> handle = Entrez.efetch(db="nucleotide",
```

```
id="57240072",rettype="genbank",email="A.N.Other@example.com")
>>> record = SeqIO.read(handle, "genbank")
>>> print record
ID: AY851612.1
Name: AY851612
Desription: Opuntia subulata rpl16 gene, intron; chloroplast.
/sequence_version=1
/source=chloroplast Austrocylindropuntia subulata
....
```

默认的，你得到的输出是 XML 格式，你可以使用 Bio.Entrez.read()函数进行分析：

```
>>> from Bio import Entrez
>>> handle = Entrez.efetch(db="nucleotide",
id="57240072",email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record[0]["GBSeq_definition"]
'Opuntia subulata rpl16 gene, intron; chloroplast'
>>> record[0]["GBSeq_source"]
'chloroplast Austrocylindropuntia subulata'
....
```

7.7 ELink

对于 ELink 的帮助，查看其帮助页面。ELink 通过 Bio.Entrez.elink()可用。

7.8 EGQuery: 获得检索条件的计数

EGQuery 提供了每一个 Entrez 数据库中，对每一个检索条件的计数。在使用 ESearch 搜索前，找出一个搜索会有多少条目返回是很有用的。在这个例子中，我们使用 Bio.Entrez.egquery()来获得包含“biopython”的数目：

```
>>> handle = Entrez.egquery(term="biopython",email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["eGQueryResult"][0]["DbName"]
'pubmed'
>>> record["eGQueryResult"][0]["Count"]
'5'
```

查看 EGQuery 帮助页面获得更多信息

7.9 ESpell: 获得拼写建议

ESpell 检索拼写建议。在这个例子中，我们使用 Bio.Entrez.espell()来获得 biopython 的正确的拼写：

```
>>> from Bio import Entrez
>>> handle = Entrez.espell(term="biopythooon", email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["Query"]
'biopythooon'
>>> record["CorrectedQuery"]
'biopython'
```

查看 ESpell 帮助页面以获得更多信息。

7.10 例子

7.10.1 搜索和下载 Entrez 核酸记录

这里，我们将展示一个远程 Entrez 检索的简单例子。在 2.3 节的分析例子中，我们讨论了使用 NCBI 的 Entrez 网站从 NCBI 核酸数据库搜索 Cypripedioideae 信息，我们的朋友--lady slipper orchids. 现在，我们将看下如何使用 python 脚本自动这个过程。在这个例子中，我们将展示使用 Entrez 模块做如何连接，获得结果，分析它们等所有工作。

首先，在下载它们之前，让我们使用 EGQuery 获得我们将要得到的结果的数目：

```
>>> from Bio import Entrez
>>> handle = Entrez.egquery(term='Cypripedioideae', email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> for row in record['eGQueryResult']:
...     if row['DbName']=='nuccore':
...         print row['Count']
814
```

这样，我们期望获得 814 条 Entrez 核酸记录。如果你发现一些高的离谱的击中项数目，你需要考虑是否要全部下载，那将是我们接下来的一步：

```
>>> from Bio import Entrez
>>> handle = Entrez.esearch(db='nucleotide', term='Cypripedioideae', retmax=814,email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
```

这里，记录是一个 python 字典，包含搜索结果和一些辅助信息。仅仅对于信息，让我们看下字典中存储的是什么：

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

首先，我们检查下我们获得了多少个记录：

```
>>> print record['Count']
'814'
```

是我们期待的数字。814 个结果被保存在 record['IdList']中:

```
>>> print len(record['IdList'])
814
```

让我们来看看前 5 个结果：

```
>>> print record['IdList'][:5]
['187237168', '187372713', '187372690', '187372688', '187372686']
```

我们可以使用 EFetch 来下载这些记录。你可以分次下载这些记录以减少载入 NCBI 的服务器；最好一次下载一系列记录，如下所示。但是，在这种情况下，你需要使用历史特性（7.10.3 节有描述）。

```
>>> idlist = ",".join(record['IdList'][:5])
>>> print idlist
187237168,187372713,187372690,187372688,187372686
>>> handle = Entrez.efetch(db='nucleotide', id=idlist, retmode='xml',email="A.N.Other@example.com")
>>> records = Entrez.read(handle)
>>> print len(records)
5
```

这些记录的每一个都个一个 GenBank 记录相关。

```
>>> 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
```

你可以使用这个快速建立搜索---但是对于大量使用，查看 7.10.3 节。

7.10.2 找出一个物种的世系

继续对相同的物种工作，现在让我们找出它的世系。首先我们对 Cypripedioideae 搜索 taxonomy 数据库. 我们发现只有一个 AC 号：

```
>>> handle = Entrez.esearch(db="Taxonomy",
term="Cypripedioideae",email="A.N.Other@example.com")
>>> record = Entrez.read(handle)
>>> record["IdList"]
['158330']
>>> record["IdList"][0]
'158330'
```

现在，我们使用 efetch 来下载 Taxonomy 数据库中的这个条目，并进行分析：

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

同样，这个记录包含很多信息

```
>>> 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']
```

我们可以从这个记录中直接获得世系：

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

7.10.3 使用历史和 WebEnv

通常，你需要进行一系列相关的检索。最典型的，运行一个搜索，也许更深的搜索，然后取回搜索结果的详情。你可以在 Entrez 中，通过构建一系列分割的调用来做这些。但是，NCBI 比较喜欢使用它们的历史支持。例如，假定我们想要搜索和下载所有 Orchid rpl16 核酸序列，存到一个 FASTA 文件中。我们可以自然的结合 Bio.Entrez.esearch()的例子代码，获得一个 GI 号的列表，然后重复调用 Bio.Entrez.efetch()以下载它们全部。你可以通过要求记录分批以减少检索的数目。那可能很好，但是仍不是 NCBI 鼓励使用的方法。被证明的方法是使用历史特性来运行搜索。然后，我们可以通过参考搜索结果取回结果- NCBI 预先进行缓存。

```
from Bio import Entrez
search_handle = Entrez.esearch(db="nucleotide",term="Opuntia and rpl16",
                               usehistory="y", email="history.user@example.com")
search_results = Entrez.read(search_handle)
search_handle.close()
gi_list = search_results["IdList"]
count = int(search_results["Count"])
assert count == len(gi_list)
session_cookie = search_results["WebEnv"]
query_key = search_results["QueryKey"]
```

除了在搜索中找到的序列的 GI 号，因为我们已经使用历史特性，XML 搜索结果也包含 WebEnv 和 QueryKey 值，用来体积这些搜索结果。以变量 session_cookie 和 query_key 存储这些值后，我们能使用它们作为 Bio.Entrez.efetch()的参数，而不是给出 GI 号作为标识符。对于小的搜索，你可能立即成功的下载所有信息，最好分批下载。你可以使用 retstart 和 retmax 参数来确定你需要返回的搜索结果的范围。(以零基计数开始，结果最大值返回)。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",
                                 retstart=start, retmax=batch_size,
                                 webenv=session_cookie, query_key=query_key,
                                 email="history.user@example.com")
    data = fetch_handle.read()
    fetch_handle.close()
    out_handle.write(data)
out_handle.close()
```

最好，不要忘记在 Entrez 调用中包括你的 email 地址。

第八章 Swiss-Prot, Prosite, Prodoc,和 ExPASy

8.1 Bio.SwissProt: 分析 Swiss-Prot 文件

8.1.1 分析 Swiss-Prot 记录

Swiss-Prot (http://www.expasy.org/sprot) 是一个人工干预的蛋白质序列数据库。在 4.2.2 节中，我们描述了，做为一个 SeqRecord 对象，如何提取 SwissProt 记录的序列。另一个选择是，你可以保存 Swiss-Prot 记录到一个 Bio.SwissProt.SProt.Record 对象，它存储了包含在 SwissProt 记录中的完整的信息。本节中，我们描述如何从一个 SwissProt 文件中提取 Bio.SwissProt.SProt.Record 对象。要分析一个 Swiss-Prot 记录，首先要得到一个 SwissProt 记录的 handle。有很多方式完成这一任务，依赖于 SwissProt 记录被存储在哪里及如何存储的：

在本地打开 SwissProt 文件

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

打开一个 gzipped 压缩格式的 Swiss-Prot 文件

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

通过互联网打开一个 Swiss-Prot 文件：

```
>>> import urllib
>>> handle = urllib.urlopen("http://www.somelocation.org/data/someswissprotfile.dat")
```

通过互联网从 ExPASy 数据库中打开一个 Swiss-Prot 文件(查看 8.4.1 节):

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

重点是对于分析器，不介意 handle 是如何创造的，只要它指出 SwissProt 格式数据。我们可以使用 4.2.2 节中描述的 Bio.SeqIO 获得文件格式（对不可知的 SeqRecord 对象？）。另外，我们可以获得 Bio.SwissProt.SProt.Record 对象，它和基本的文件格式几乎匹配，使用以下代码。从一个 handle 中读取 Swiss-Prot 记录，我们可以使用 read()函数：

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

如果 handle 指向刚好一个 SwissProt 记录，这个函数可以被使用。如果没有找到 SwissProt 记录或记录多于一个，则会引发一个 ValueError。我们可以打印出关于这个记录的一些信息：

```
>>> print record.description
CHALCONE SYNTHASE 3 (EC 2.3.1.74) (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', 'Embryophyta', 'Tracheophyta', 'Spermatophyta','Magnoliophyta', 'Liliopsida', 'Asparagales', 'Orchidaceae', 'Bromheadia']
```

要分析包含多于一个 SwissProt 记录的文件，我们可以使用分析函数来代替。这个函数允许我们对文件中的记录进行重复。例如，让我们分析下全 SwissProt 数据库，收集全部的描述。所有的 SwissProt 数据库，包含 290484 个记录：（一个 uniprot_sprot.dat.gz 压缩文件）

(linux 上使用 gzip)

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

```
290484
>>> descriptions[:3]
['104 kDa microneme/rhoptry antigen precursor (p104).',
 '104 kDa microneme/rhoptry antigen precursor (p104).',
 'Protein 108 precursor.']
```

很容易从 SwissProt 记录中提取你想要的任何种类的信息。要查看 Swiss-Prot 记录成员，使用：

```
>>> 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']
```

8.1.2 分析 Swiss-Prot 关键词和类别列表

Swiss-Prot 也提供了一个叫做 keywlist.txt 的文件，列出了在 SwissProt 中使用的关键词和类别。文件包含的条目如下面的形式：

```
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.
...

这个文件中的条目可以使用 Bio.SwissProt.KeyWList 模块中的分析函数来分析。每一个条目作为一个 python 字典类型，都被存储成一个 Bio.SwissProt.KeyWList.Record。

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

这会打印出：

```
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.
...
```

8.2Bio.Prosite：分析 Prosite 记录

Prosite 是一个包含蛋白质域，蛋白质家族，功能位点及识别它们的模型和 profile 的一个数据库。Prosite 是和 Swiss-Prot 平行开发的. 在 Biopython 中，一个 Prosite 记录通过 Bio.Prosite.Record 类来代表，它的成员涉及到 Prosite 记录的不同领域。一般的讲，一个 Prosite 文件可以包含多于一个 Prosite 记录。例如，成套的 Prosite 记录可以从 ExPASy 上下载(作为一个单文件 prosite.dat)，它包含有 2073 个记录(2007 年 12 月 4 日的 20.24 版本)。为分析这个文件，我们仍需要使用一个 iterator：

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

现在我们可以一次提取一个记录，打印出一些信息。例如，使用包含全部 Prosite 数据库的文件，我们会发现：

```
>>> from Bio 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'
```

等等。

如果你对有多少 Prosite 记录感兴趣，可以使用：

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

```
>>> print n
2073
```

从 handle 中读取一个 Prostite，你可以使用 read 函数：

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

如果没有发现 Prostite 记录，或者发现多于一个 Prosite 记录，该函数会引发一个 ValueError。

8.3 Bio.Prosite.Prodoc：分析 Prodoc 记录

在上面的 Prosite 例子中, record.pdoc 的 AC 号 'PDOC00001', 'PDOC00004', 'PDOC00005' 等指的是 Prodoc 记录，包含 Prosite 文档。Prodoc 记录可以作为从 ExPASy 得来的有用的单文件，或者作为一个包含所有 Prodoc 记录的一个文件。我们使用 Bio.Prosite.Prodoc 中的分析器来分析 Prodoc 记录。例如，要创建一个所有 ProdocAC 号的列表，可以使用：

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

read()函数被提供用来从 handle 中读取一个 Prodoc 记录。

8.4Bio.ExPASy：访问 ExPASy 服务器

Swiss-Prot, Prosite, 及 Prodoc 记录可以从 ExPASy 网络服务器(http://www.expasy.org/)上下载。六类检索是可行的：

get_prodoc_entry 以 HTML 格式下载 Prodoc 记录

get_prosite_entry 以 HTML 格式下载 Prosite 记录

get_prosite_raw 以原始格式下载 Prosite 或 Prodoc 记录

get_sprot_raw 以原始格式下载 SwissProt 记录

sprot_search_ful 检索 SwissProt 记录

sprot_search_de 检索 SwissProt 记录

从 python 脚本来访问这个网络服务器，我们可以使用 Bio.ExPASy 模块

8.4.1 检索一个 SwissProt 记录

比如说你在查找 Orchids 的 chalcone 合酶。Chalcone 合酶涉及到植物的黄烷类生物合成，黄烷类物质很有用，例如色素颜色及 UV 杀虫剂等。如果你在 SwissProt 上做一个搜索，你会找到 Chalcone 合酶的三个 orchid 蛋白质，ID 号分别是：O23729, O23730, O23731。现在，让我们写一个脚本来攫取这些内容，分析出一些有用的信息。首先，我们攫取记录，使用 Bio.ExPASy 中的 get_sprot_raw() 函数。这个函数很好，因为你可以提供一个 ID，然后得到一个未加工的 text 记录的一个 handle（没有 html 来捣乱）。然后，我们可以使用 Bio.SwissProt.read 来抽出 SwissProt 记录，或者使用 Bio.SeqIO.read 得到一个 SeqRecord。以下代码实现了我刚才所写的：

```
>>> 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 AC 号不存在，那样 SwissProt.read(handle) 就会引发一个 ValueError。你可以抓取 ValueException 异常以探测无效的 AC 号：

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

8.4.2 搜索 Swiss-Prot

现在，你可能注意到我事先知道记录的 AC 号。当然，get_sprot_raw()需要入口名或者一个 AC 号。当你手边没有它们时，你可以使用 sprot_search_de() 或者 sprot_search_ful()函数。sprot_search_de() 搜索 ID, DE, GN, OS 及 OG 行；

sprot_search_ful() 搜索几乎所有领域。查看它们的详细说明：http://www.expasy.org/cgi-bin/sprot-search-de 及 http://www.expasy.org/cgi-bin/sprot-search-ful。注意它们默认是不搜索 TrEMBL 的，它们返回 htmal 页面，但是 AC 号很容易提取：

```
>>> 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)
```

8.4.3 检索 Prosite 和 Prodoc 记录

Prosite 和 Prodoc 记录可以用 HTML 格式或原始格式进行检索。使用 biopython 来分析 Prosite 和 Prodoc 记录，你需要以原始格式检索这些记录。但是，对于其他目的，你可能对这些记录的 HTML 格式感兴趣。使用 get_prosite_raw() 来检索 raw 格式的 Prosite 或 Prodoc 记录。尽管这个函数名字中有 prosite，它也可以对 Prodoc 记录使用。例如，下载 Prosite 记录，以 raw 文本格式打印，使用：

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

检索一个 Prosite 记录，把它分析成 Bio.Prosite.Record 对象，使用：

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

最后，检索一个 Prodoc 记录，把它分析成 Bio.Prosite.Prodoc.Record 对象，使用:

```
>>> from Bio import ExPASy
>>> from Bio.Prosite import Prodoc
>>> handle = ExPASy.get_prosite_raw('PDOC00001')
```

```
>>> record = Prodoc.read(handle)
```

对于不存在的 AC 号，ExPASy.get_prosite_raw 返回一个空字符串的 handle。当面对一个空字符串时，Prosite.read 及 Prodoc.read 将会引发一个 ValueError。你可以获取这些例外来检查无效的 AC 号。get_prosite_entry()及 get_prodoc_entry()被用来下载 HTML 格式的 prosite 和 prodoc 记录。要创建一个 web 页面显示一个 prosite 记录，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()
```

对于 prodoc 记录也很相似：

```
>>> 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，一个无效的 AC 号将返回一个 html 格式的错误信息。

第九章 Cookbook –使用它来做些酷事情

9.1 PubMed

Bio.PubMed 模块使用 Bio.Entrez 从内部查询 NCBI.请记着阅读和研究在线使用 NCBI 的 Entrez 指南。如果你被发现滥用它们的服务器，它们会锁定你的查询权限。参考 7.1 节。

9.1.1 向 PubMed 提交一个检索

如果你在进行医学研究或者对人类组织感兴趣，PubMed 绝对是一个很好的资源。像其他事情一样，我们可以从它获取信息，在 python 脚本中使用它。使用 biopython 进行 PubMed 检索是很轻松的。让我们来获得所有关于 orchids 的文献的 ID。我们仅仅需要以下三行：

```
from Bio import PubMed
search_term = 'orchid'
orchid_ids = PubMed.search_for(search_term)
```

将返回一个包含所有 orchidID 的 python 列表：
['11070358', '11064040', '11028023', '10947239', '10938351', '10716342',...]
有很多，故省略。有了这些 ID 的列表，我们可以检索记录，继续下面的一节。

9.1.2 检索 PubMed 记录

前一节描述了如何得到一系列的文献 ID。既然我们已经得到它们，显然，我们想得到对应的 Medline 记录，然后从中提取信息。从 PubMed 中提取记录的界面对于 python 编程者来说是很直观的--它模拟一个 python 的字典。建立这个界面，我们需要建立一个分析器以分析我们检索到的结果
。下面的几行代码可以建立所有的东西：

```
from Bio import PubMed
from Bio import Medline
rec_parser = Medline.RecordParser()
medline_dict = PubMed.Dictionary(parser = rec_parser)
```

我们已经创建了一个 medline_dict 对象的字典。像使用字典一样来获得文献，如 medline_dict[id_to_get]。这个所做的就是连接 PubMed，得到
你需要的文献，把它分析成一个记录对象，然后返回。很酷！现在让我们来看看如何使用这个美好的字典来打印出关于某些 ID 的一些信息。我
们仅仅需要对 ID（前一节得到的）进行循环，然后打印出我们感兴趣的信息：

```
for oid in orchid_ids[0:5]:
    cur_record = medline_dict[oid]
    print 'title:', cur_record.title.rstrip()
    print 'authors:', cur_record.authors
    print 'source:', cur_record.source.strip()
    print
```

输出是像这样的：

```
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
```

需要注意的特别有趣的是作者的列表，以标准 python 列表形式返回。这使得它很容易使用标准 python 工具进行操控及检索。例如，我们可以使
用下面的代码，通过一个整体的条目循环来检索一个特定的作者：

```
search_author = 'Waits T'
for our_id in our_id_list:
    cur_record = medline_dict[our_id]

    if search_author in cur_record.authors:
       print "Author %s found: %s" % (search_author,cur_record.source.strip())
```

PubMed 和 Medline 界面很成熟，也很易用--希望本节给了你一个关于界面和如何使用的很好的想法。

9.2GenBank

GenBank 记录格式是一个很流行的方法来存储序列信息，序列特征，以及其他相关序列信息。这个格式是一个从 NCBI 数据库中获取信息的很好的
方式。Bio.GenBank 模块使用 NCBI 的 Entrez 界面来进行搜索和提取数据。请记

着阅读和研究在线使用 Entrez 的 NCBI 的指南。如果你被发现滥用它们的服务器，它们会锁定你的查询权限。参考 7.1 节。

9.2.1 从 NCBI 检索 GenBank 条目

GenBank 库的一个很好的特性是可以从 GenBank 中自动检索输入的能力。对于创建脚本自动做很多日常工作来说，这是很方便的。在这个例子中，我们将展示如何检索 NCBI 数据库，从查询项检索记录--在 4.2.1 中涉及到的。首先，我们需要建立一个检索项，找出检索记录的 ID。这里，我们将快速检索我们最喜欢的物种--Opuntia.我们可以做一个快速检索，得到所有相关的返回的 GI 号：

```
from Bio import GenBank
gi_list = GenBank.search_for("Opuntia AND rpl16")
```

gi_list 将会是一个击中项的有关 GI 的一个列表：

```
["6273291", "6273290", "6273289", "6273287", "6273286", "6273285", "6273284"]
```

既然我们得到了 GI 号，我们可以使用它们通过字典界面来检索 NCBI 数据库。例如，要检索第一个 GI 的信息，我们首先需要创建一个字典以存储 NCBI：

```
ncbi_dict = GenBank.NCBIDictionary("nucleotide", "genbank")
```

我们可以进行检索：

```
gb_record = ncbi_dict[gi_list[0]]
```

在这个例子中，gb_record 会是 GenBank 格式记录形式：

```
LOCUS       AF191665      902 bp    DNA             PLN       07-NOV-1999
DEFINITION  Opuntia marenae rpl16 gene; chloroplast gene for chloroplast
            product, partial intron sequence.
ACCESSION   AF191665
VERSION     AF191665.1  GI:6273291
...
```

这个例子中，我们仅仅得到了原始记录，我们也可以把这些记录直接传递到一个分析器中，返回分析过的记录。例如，如果你需要得到由 GenBank 文件分析成 SeqFeature 对象的 SeqRecord 对象，我们需要创建一个包含 GenBank FeatureParser 的字典：

```
record_parser = GenBank.FeatureParser()
ncbi_dict = GenBank.NCBIDictionary("nucleotide", "genbank",parser =
record_parser)
```

现在，检索记录将返回一个 SeqRecord 记录，而不是原始记录：

```
>>> gb_seqrecord = ncbi_dict[gi_list[0]]
>>> print gb_seqrecord
<Bio.SeqRecord.SeqRecord instance at 0x102f9404>
```

使用这些自动检索函数是做这些事的一个很好的补充。尽管这个模块需要遵守 NCBI 的 3-2 规则，NCBI 还有其他的建议，例如避免高峰期等。参考 7.1 节。分析 GenBank 记录信息的更多格式信息，请查看 9.2.2 节。