BIOINFORMATICS · PYTHON

Biopython
零基础实战教程

从零开始学会用 Python 处理生物序列

示例代码基于 Biopython 1.86 环境全部验证通过

适合完全零基础的生物信息学初学者

2026 年 8 月

目　录

前言 6

本书约定 6

环境要求 6

第 1 章 认识 Biopython 7

1.1 什么是 Biopython 7

1.2 为什么要学 Biopython 7

1.3 Biopython 模块地图 7

1.4 本书学习路线 8

第 2 章 环境搭建与安装 10

2.1 安装 Python 10

2.2 安装 Biopython 10

2.3 验证安装 11

2.4 推荐开发环境 11

2.5 安装其他常用配套库 11

第 3 章 序列对象 Seq —— 一切从这里开始 12

3.1 创建 Seq 对象 12

3.2 基本操作：长度、索引、切片 12

3.3 序列运算：拼接与重复 13

3.4 互补与反向互补（DNA 双链的核心操作） 13

3.5 转录与翻译（中心法则） 14

3.6 Seq 的"身份"：比较与哈希 15

3.7 小结与练习 15

第 4 章 SeqRecord 与 SeqIO —— 读写生物序列文件 17

4.1 先认识 FASTA 格式 17

4.2 SeqRecord：带注释的序列 17

4.3 用 SeqIO.parse() 读取多条记录 18

4.4 用 SeqIO.read() 读取单条记录 19

4.5 用 SeqIO.write() 写出文件 19

4.6 格式转换：FASTA ↔ GenBank 20

4.7 批量处理：过滤与筛选 20

4.8 大文件：用迭代器节省内存 21
4.9 小结与练习 22

第 5 章 序列分析工具箱 Bio.SeqUtils 23

5.1 GC 含量：序列的最基本统计量 23
5.2 序列的分子量 23
5.3 蛋白质理化性质分析 ProtParam 23
5.4 在序列中查找子序列 24
5.5 综合示例：批量计算 GC 含量并统计 25
5.6 小结与练习 25

第 6 章 序列比对 27

6.1 一个直观的例子 27
6.2 PairwiseAligner：双序列比对 27
6.3 局部比对与参数调节 28
6.4 多序列比对对象 MultipleSeqAlignment 29
6.5 AlignIO：读写比对文件 29
6.6 实战场景：从比对中找保守位点 30
6.7 小结与练习 31

第 7 章 在线数据库与 BLAST 32

7.1 设置邮箱与获取 NCBI 密钥 32
7.2 搜索数据库：Entrez.esearch 32
7.3 下载序列：Entrez.efetch 33
7.4 BLAST 比对：找相似序列 33
7.5 PubMed 文献检索 34
7.6 小结与练习 35

第 8 章 蛋白质三维结构 Bio.PDB 36

8.1 PDB 文件长什么样 36
8.2 用 PDBParser 读取结构 36
8.3 常用操作 37
8.4 蛋白结构可视化 38
8.5 小结与练习 38

第 9 章 系统发育分析 Bio.Phylo 40

9.1 树文件格式：Newick 40
9.2 读取与遍历树 40

9.3 从多序列比对构建进化树 40
9.4 美化绘制进化树 42
9.5 树的常用操作 42
9.6 小结与练习 43

第 10 章 其他实用模块 44

10.1 限制性内切酶分析 Bio.Restriction 44
10.2 序列模体 Bio.motifs 44
10.3 密码子使用偏好 45
10.4 其他值得一提的模块 46

第 11 章 综合实战：三个完整的项目 47

11.1 项目一：基因序列分析报告生成器 47
11.2 项目二：序列清洗与格式整理工具 48
11.3 项目三：ORF 查找器 49
11.4 项目四：批量蛋白理化性质分析 51

第 12 章 调试技巧与常见错误 52

12.1 高频错误速查表 52
12.2 字符串与 Seq 的转换陷阱 52
12.3 中文与编码问题 53
12.4 网络请求失败处理 54
12.5 调试三件套 55

第 13 章 结语与进阶路线 56

13.1 官方资源 56
13.2 下一步学习路线 56
13.3 学习建议（来自实践） 56

附录 A：常用代码速查表 58

A.1 序列操作 58
A.2 文件读写 58
A.3 分析工具 59
A.4 数据库与 BLAST 59
A.5 进化树 59
A.6 蛋白质结构 60

附录 B：练习参考答案 61

第 3 章 61

第 4 章 61

第 5 章 62

第 6 章 63

第 7 章 63

第 9 章 64

第 11 章 64

前言

《**Biopython 零基础实战教程**》 是一本面向完全零基础读者的生物信息学入门书。你不需要任何生物学编程背景，只需要会打开命令行、会运行 Python 脚本，就可以跟着本书一步步学会用 Biopython 处理 DNA、RNA 和蛋白质序列。

本书的每一个代码示例都经过 Biopython 1.86 环境实际运行验证，你可以放心复制运行。全书采用"先概念、后代码、再练习"的编排方式，每章末尾都配有思考题，附录 B 提供了参考答案。

> **如何使用本书** 建议你按照章节顺序阅读，并**亲自动手敲一遍代码**——编程是"手上功夫"，只看不练等于没学。遇到不理解的概念，先跳过，读到后面往往就豁然开朗了。

本书约定

标记	含义
代码块	Python 代码、命令或示例输出
加粗	重点概念或需要记住的术语
> 引用框	提示、注意事项或背景知识
注意：	容易踩坑的地方，请特别留意
提示：	实用小技巧

环境要求

- Python 3.9 及以上版本（推荐 3.11+）
- Biopython 1.80 及以上版本（本书示例基于 1.86 验证）
- 网络连接（第 7 章访问 NCBI 数据库时需要）

第 1 章 认识 Biopython

1.1 什么是 Biopython

Biopython 是一套开源的 Python 库，专门用于生物学计算。它由全球开发者社区维护，是生物信息学领域最流行的 Python 工具包之一，2000 年首次发布，如今已迭代到 1.8x 版本。

它解决了生物学研究中的一个核心痛点：**生物数据格式繁多**。DNA 序列有 FASTA、GenBank、EMBL 格式，蛋白结构有 PDB 格式，比对结果有 Clustal、Stockholm 格式，进化树有 Newick 格式……如果没有统一工具，每换一种格式就要重写一遍解析代码。Biopython 把这些格式的读写统一封装成简单的 Python 接口，让你专注于**生物学问题本身**，而不是文件解析。

> **一句话理解**：Biopython 之于生物学家，就像 `requests` 之于爬虫工程师、`pandas` 之于数据分析师——把脏活累活都包了，你只管用。

1.2 为什么要学 Biopython

- **零门槛上手**：只需要基础 Python 语法（变量、列表、循环、函数）即可开始；
- **生态完善**：序列分析、数据库检索、比对、进化树、蛋白结构全覆盖；
- **社区庞大**：Stack Overflow 上有海量问答，官方文档详尽，遇到问题容易搜到答案；
- **与科研流程无缝衔接**：测序公司交付的 FASTQ、NCBI 下载的 GenBank、AlphaFold 输出的 PDB，都能直接读取；
- **可扩展**：和 NumPy、pandas、matplotlib 完美配合，能做统计分析和可视化。

1.3 Biopython 模块地图

Biopython 不是一个大而全的"黑盒"，而是由若干子模块组成的工具箱。下面这张表是本书的"地图"，也是你日后查阅文档的索引：

子模块	功能	对应章节
Bio.Seq	序列对象，DNA/RNA/蛋白的存储与操作	第 3 章
Bio.SeqRecord	带注释（ID、描述、特征）的序列记录	第 4 章
Bio.SeqIO	序列文件的读写与格式转换	第 4 章
Bio.SeqUtils	GC 含量、分子量等序列分析工具	第 5 章
Bio.Align	双序列比对（PairwiseAligner）	第 6 章
Bio.AlignIO	多序列比对文件的读写	第 6 章
Bio.Entrez	访问 NCBI 数据库（搜索、下载）	第 7 章
Bio.Blast	运行与解析 BLAST 比对	第 7 章
Bio.PDB	蛋白质三维结构解析	第 8 章
Bio.Phylo	系统发育树分析与可视化	第 9 章
Bio.Restriction	限制性内切酶分析	第 10 章
Bio.motifs	序列模体（motif）分析	第 10 章
Bio.SeqUtils.ProtParam	蛋白质理化性质计算	第 5 章

1.4 本书学习路线

```
第 1~2 章  打基础：认识 Biopython，装好环境
    ↓
第 3 章    学核心：Seq 序列对象
    ↓
第 4 章    学读写：SeqRecord + SeqIO（最常用的组合）
    ↓
第 5 章    学分析：GC 含量、蛋白性质
    ↓
第 6~9 章  进阶：比对、数据库、结构、进化树
    ↓
第 10~11 章 实战：模块串烧 + 完整项目
    ↓
第 12 章   排错：常见错误与调试
```

> 提示： **学习建议**：第 3~5 章是"地基"，务必吃透；第 6~9 章按需学习，用到哪章读哪章；第 11 章的实战项目是对全书最好的检验。

第 2 章 环境搭建与安装

2.1 安装 Python

如果你还没有 Python，请到 python.org 下载 3.11 或 3.12 版本安装。

- **Windows**：安装时务必勾选 **"Add Python to PATH"**；
- **macOS**：推荐用 Homebrew 安装：`brew install python`；
- **Linux**：多数发行版自带 Python，用系统包管理器安装即可。

安装完成后，在终端（命令提示符）里验证：

```
python --version
```
终端

> 提示：如果你不习惯命令行，可以直接安装 **Anaconda**（自带 Python 和大量科学计算包），用 Anaconda Prompt 操作。

2.2 安装 Biopython

方法一：pip（推荐）

```
pip install biopython
```
终端

如果系统提示权限问题，加 `--user` 参数，或使用虚拟环境：

```
python -m venv bioenv        # 创建虚拟环境
source bioenv/bin/activate   # Windows 用 bioenv\Scripts\activate
pip install biopython
```
终端

方法二：conda

```
conda install -c conda-forge biopython
```
终端

2.3 验证安装

打开 Python 交互式环境（终端输入 `python`），输入：

```python
import Bio
print(Bio.__version__)
```

如果输出类似 `1.86` 的版本号，说明安装成功。

> 注意： 安装失败怎么办？最常见的原因是网络问题。国内用户可以换用清华镜像源： `pip install biopython -i https://pypi.tuna.tsinghua.edu.cn/simple`

2.4 推荐开发环境

- **Jupyter Notebook / Jupyter Lab**：适合边写边看结果，生物信息学入门首选。安装：`pip install jupyter`，启动：`jupyter notebook`；
- **VS Code**：安装 Python 扩展后即可获得代码补全和调试功能，适合写正式脚本；
- **交互式测试**：任何时候都可以在 Python 终端里快速验证一行代码的效果。

> 提示： 本书所有示例都可在 Jupyter 中逐行运行，也能存成 `.py` 脚本用 `python 文件名.py` 执行。

2.5 安装其他常用配套库

Biopython 经常与以下库搭配使用，建议一并安装：

```
pip install numpy matplotlib scipy
```

- `numpy`：数值计算，Biopython 的矩阵、距离计算会用到；
- `matplotlib`：绘图，绘制进化树、比对图、GC 分布图；
- `scipy`：科学计算，部分统计功能依赖它。

> 完成： **环境检查清单** 1. `python --version` 能输出版本号； 2. `import Bio` 不报错； 3. `Bio.__version__` 大于等于 1.80； 4. `import numpy, matplotlib` 正常。 全部通过就可以开始第 3 章了！

第 3 章 序列对象 Seq —— 一切从这里开始

生物信息学处理的核心数据是**序列**：DNA 由 A、T、G、C 四种碱基组成，RNA 把 T 换成了 U，蛋白质由 20 种氨基酸组成。Biopython 用 `Seq` 对象来代表一条序列——它是整个库的基石，几乎一切功能都围绕它展开。

3.1 创建 Seq 对象

```python
from Bio.Seq import Seq

# 创建一条 DNA 序列
dna = Seq("ATGCATGC")
print(dna)          # ATGCATGC
print(type(dna))    # <class 'Bio.Seq.Seq'>

# 创建 RNA 和蛋白质序列
rna = Seq("AUGCAUGC")
protein = Seq("MAIVMGR")
```

Seq 对象本质上是一段**带字母表（alphabet）标记**的字符串，但它比普通字符串多了大量生物学方法。

> 注意：**大小写敏感**：Biopython 默认不区分碱基大小写，但为了可读性，惯例上 DNA/RNA 用大写。注意 `Bio.Seq.Seq` 与 `Bio.SeqRecord.SeqRecord` 是两个不同的类，前者是"裸序列"，后者是"带注释的序列"（第 4 章详解）。

3.2 基本操作：长度、索引、切片

Seq 支持所有字符串操作：

```python
seq = Seq("ATGCATGC")

print(len(seq))          # 8 —— 序列长度
print(seq[0])            # A —— 第一个碱基（索引从 0 开始）
print(seq[-1])           # C —— 最后一个碱基
print(seq[2:5])          # GCA —— 切片，取第 3~5 个碱基
print(seq[::2])          # ATAT —— 隔一个取一个
```

3.3 序列运算：拼接与重复

```python
seq1 = Seq("ATGC")
seq2 = Seq("TGCA")

# 拼接两条序列
combined = seq1 + seq2
print(combined)          # ATGCTGCA

# 重复序列
print(seq1 * 2)          # ATGCATGC

# 判断子序列是否在序列中
print("ATG" in seq1)     # True

# 统计碱基出现次数
print(seq1.count("A"))   # 1
```

> 提示： `count` 和 `len` 都支持，注意顺序： `seq.count("A")` 而不是 `"A".count(seq)`。

3.4 互补与反向互补（DNA 双链的核心操作）

DNA 双链中 A 配 T、G 配 C。互补（complement）和反向互补（reverse complement）是分子生物学最常用的操作——比如设计引物、判断基因所在链时都要用到。

```python
seq = Seq("ATGCATGC")

# 互补链
print(seq.complement())          # TACGTACG

# 反向互补（常用！）
print(seq.reverse_complement())  # GCATGCAT

# 对 RNA 同样适用（A 配 U）
rna = Seq("AUGCAUGC")
print(rna.complement())          # UACGUACG
```

> 注意：**千万别把二者搞混**：`complement()` 只是把每个碱基换成配对碱基；`reverse_complement()` 是先把序列**倒过来**再互补——这是双链 DNA 另一条链的真实方向，实验和数据分析中用得最多。

3.5 转录与翻译（中心法则）

分子生物学的中心法则：DNA → RNA（转录）→ 蛋白质（翻译）。Seq 对象内置了这两个操作：

```python
dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")

# 转录：DNA -> mRNA（T 变成 U）
mrna = dna.transcribe()
print(mrna)   # AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG

# 翻译：mRNA -> 蛋白质（每 3 个碱基一个密码子）
protein = mrna.translate()
print(protein)   # MAIVMGR*KGAR*

# 也可以直接从 DNA 翻译（等价）
print(dna.translate())   # MAIVMGR*KGAR*
```

输出中的 `*` 是**终止密码子**（Stop codon）。如果不希望显示，可以加参数：

```python
# 翻译遇到第一个终止密码子即停止
print(dna.translate(to_stop=True))    # MAIVMGR

# 如果序列不是 3 的倍数，默认会丢弃末尾不完整的密码子
print(dna.translate())                # MAIVMGR*KGAR*
```

> 提示：**翻译的方向**：`translate()` 默认从第 0 位开始（读框 1）。要读其他读框，先切片再翻译，例如 `dna[1:].translate()` 就是读框 2。第 11 章的 ORF 查找器会用到这个技巧。

3.6 Seq 的"身份"：比较与哈希

Seq 支持相等比较，也支持作为字典的键：

```python
seq_a = Seq("ATGC")
seq_b = Seq("ATGC")
seq_c = Seq("ATGG")

print(seq_a == seq_b)   # True —— 序列内容相同
print(seq_a == seq_c)   # False

# 字典键（用于去重、统计）
counter = {}
counter[seq_a] = counter.get(seq_a, 0) + 1
print(counter)          # {Seq('ATGC'): 1}
```

3.7 小结与练习

本章核心速记：

操作	方法/语法	示例结果
创建	`Seq("ATGC")`	`Seq('ATGC')`
长度	`len(seq)`	`4`
拼接	`seq1 + seq2`	`ATGC...`
互补	`seq.complement()`	`TACG`
反向互补	`seq.reverse_complement()`	`GCAT`
转录	`seq.transcribe()`	`AUGC`
翻译	`seq.translate()`	`M...`

练习：

1. 创建序列 `"AATTCCGG"`，输出它的互补序列和反向互补序列。
2. 有一条 DNA：`"ATGGTGGCATGA"`，先转录再翻译，输出蛋白质序列（提示：用 `to_stop=True`）。
3. 写一行代码，统计 `"ACGTACGTACGT"` 中每个碱基各出现几次（提示：结合 `count` 和循环，或用 `collections.Counter`）。

> 提示： 答案见附录 B。卡住了先自己思考，实在不行再看答案。

第 4 章 SeqRecord 与 SeqIO —— 读写生物序列文件

第 3 章的 `Seq` 对象是"裸序列"，但现实中的序列数据总是带着一大堆注释：基因 ID、物种名、功能描述、编码区位置……Biopython 用 **`SeqRecord`** 承载这些信息，用 **`SeqIO`** 负责文件的读写。这三个类加起来，是 Biopython 日常使用率最高的组合。

4.1 先认识 FASTA 格式

FASTA 是生物信息学最通用的序列文本格式，**每条记录由两行组成**：

```
>基因ID 描述信息（可有可无）
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
```

- 以 `>` 开头的行为**标题行**，`>` 后紧跟的是序列 ID，后面的空格及之后的内容是描述；
- 之后可以有一行或多行序列（通常每行 60~80 个字符，纯文本，无空格）；
- 一个 FASTA 文件可以包含任意多条记录。

4.2 SeqRecord：带注释的序列

```python
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

record = SeqRecord(
    Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG"),  # 序列本体
    id="gene_001",           # 唯一标识（对应 FASTA 中 > 后面的第一个词）
    name="demo_gene",        # 名称
    description="a demo gene from tutorial",  # 描述
)

print(record.id)             # gene_001
print(record.description)    # a demo gene from tutorial
print(record.seq)            # ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
print(len(record))           # 42 —— 注意 len() 作用于整个记录时返回序列长度
```

`SeqRecord` 的常用属性：

属性	含义	示例
.seq	序列（Seq 对象）	record.seq
.id	唯一标识	"gene_001"
.name	名称	"demo_gene"
.description	描述	"a demo gene"
.features	特征列表（基因、CDS 等注释）	第 4.6 节
.annotations	其他元数据（字典）	物种、数据库来源等

4.3 用 SeqIO.parse() 读取多条记录

`SeqIO.parse(文件, 格式)` 返回一个**迭代器**，逐个产出记录。它是读取序列文件最核心的函数：

```python
from Bio import SeqIO

# 方法一：直接传文件路径
records = SeqIO.parse("genes.fasta", "fasta")
for record in records:
    print(record.id, len(record.seq))

# 方法二：用 with 语句打开文件（推荐，自动关闭文件）
with open("genes.fasta") as handle:
    for record in SeqIO.parse(handle, "fasta"):
        print(record.id, record.description)

# 方法三：一次性读入列表（适合小文件）
records = list(SeqIO.parse("genes.fasta", "fasta"))
print(len(records))   # 文件中有几条记录
```

> 注意：**parse 返回的是迭代器**：它按需读取文件，内存占用小，但**只能遍历一次**。如果遍历完还要再用，请先用 `list()` 转成列表，或者重新 parse。

4.4 用 SeqIO.read() 读取单条记录

当文件**恰好只有一条**记录时，用 `read()` 更简洁。如果文件里有多条，`read()` 会直接报错——这其实是好事，能帮你发现数据问题：

```python
from Bio import SeqIO

record = SeqIO.read("single_gene.fasta", "fasta")
print(record.id, record.seq)
```

> 注意： `SeqIO.read` 遇到多条记录会抛出 `ValueError`； `SeqIO.parse` 遇到格式错误会抛出解析异常。读文件时做好异常处理是好习惯（第 12 章详述）。

4.5 用 SeqIO.write() 写出文件

```python
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

records = [
    SeqRecord(Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG"), id="g1", description="gene one"),
    SeqRecord(Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATTAG"), id="g2", description="gene two"),
]

# 写入 FASTA 文件（格式由文件名后缀和 format 参数共同决定）
SeqIO.write(records, "output.fasta", "fasta")
```

输出到文件后内容如下：

```
>g1 gene one
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
>g2 gene two
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATTAG
```

> 提示： `SeqIO.write` 接收**列表或迭代器**作为第一个参数。注意： `parse` 返回的迭代器在用完后不能再传给 `write` ——所以要重新 parse 一次。

4.6 格式转换：FASTA ↔ GenBank

`SeqIO` 最爽的功能之一：**同一段代码，换个格式名，就能做格式转换**。GenBank 是 NCBI 最常用的注释格式，包含基因位置、编码区、CDS 等丰富信息：

```python
from Bio import SeqIO

# 把 FASTA 转成 GenBank（GenBank 需要有 features 才有意义，这里演示流程）
records = list(SeqIO.parse("genes.fasta", "fasta"))
SeqIO.write(records, "genes.gb", "genbank")

# 反向：GenBank 转 FASTA
records = list(SeqIO.parse("genes.gb", "genbank"))
SeqIO.write(records, "back_to_fasta.fasta", "fasta")
```

读取一份真实的 GenBank 文件（比如从 NCBI 下载的），可以看到 `.features` 里的注释信息：

```python
record = SeqIO.read("NC_000913.gb", "genbank")   # 大肠杆菌基因组示例
print(record.id)                # NC_000913.3
print(record.annotations["organism"])  # 物种名
print(len(record.features))     # 特征数量（基因、CDS、tRNA 等）
print(record.features[0])       # 第一条特征
```

> 提示： 从 NCBI 下载 GenBank 文件：打开 www.ncbi.nlm.nih.gov，搜索基因或基因组 → 选择 "GenBank" 格式 → 下载保存即可。第 7 章会教你用代码直接下载。

4.7 批量处理：过滤与筛选

实际工作中最常见的场景：读入一大批序列，过滤筛选，再写出。比如"只保留长度超过 100 的序列"：

```python
from Bio import SeqIO

def filter_sequences(input_file, output_file, min_length=100):
    """读取 FASTA，过滤长度不足的序列，写出新文件"""
    kept = []
    for record in SeqIO.parse(input_file, "fasta"):
        if len(record.seq) >= min_length:
            kept.append(record)
    SeqIO.write(kept, output_file, "fasta")
    print(f"原始记录：已过滤，保留 {len(kept)} 条")

filter_sequences("all_genes.fasta", "long_genes.fasta", min_length=100)
```

4.8 大文件：用迭代器节省内存

`SeqIO.parse` 返回迭代器意味着你**不必把整个文件读进内存**——处理 GB 级的测序数据时，这是生死攸关的能力。下面这段代码逐条处理几十万条序列也不会内存爆炸：

```python
from Bio import SeqIO

total_gc = 0
total_len = 0
count = 0

for record in SeqIO.parse("huge_file.fasta", "fasta"):
    seq = record.seq
    total_gc += seq.count("G") + seq.count("C")
    total_len += len(seq)
    count += 1

print(f"共 {count} 条序列，总长度 {total_len}，平均 GC 含量 {total_gc/total_len:.1%}")
```

> 提示：**什么时候用 list()，什么时候用迭代器？** 文件小（几千条以内）、需要反复访问 → 用 `list()` 方便；文件大、只需顺序处理 → 用迭代器省内存。这是面试和实际工作中都常被问到的问题。

4.9 小结与练习

本章核心 API 速记：

函数	作用	返回
`SeqIO.parse(文件, 格式)`	读取多条记录	迭代器
`SeqIO.read(文件, 格式)`	读取单条记录	SeqRecord
`SeqIO.write(记录列表, 文件, 格式)`	写出记录	写入条数
`SeqRecord(seq, id=..., description=...)`	构造带注释序列	SeqRecord

练习：

1. 用 `SeqIO.parse` 读取任意 FASTA 文件，打印每条记录的 `id` 和长度。
2. 编写代码：读取 `all.fasta`，把所有**含终止密码子早停（翻译后以 * 结尾）的序列**剔除……提示：先翻译，检查蛋白质序列中是否含 `*`（排除末尾的终止符）。
3. 把练习 1 中的文件转换成 GenBank 格式，再转换回来，验证序列内容是否一致（提示：比对转换前后的序列列表是否相等）。

第 5 章 序列分析工具箱 Bio.SeqUtils

掌握了序列的读写之后，我们开始真正"分析"序列。`Bio.SeqUtils` 提供了一大批实用的序列分析函数，本节介绍最常用的几个。

5.1 GC 含量：序列的最基本统计量

GC 含量（G 和 C 碱基占全部碱基的比例）是衡量 DNA 稳定性和物种特性的基本指标——GC 含量高的序列更耐热，不同物种的基因组 GC 含量差异很大（人 ~41%，大肠杆菌 ~50%）。

```python
from Bio.SeqUtils import gc_fraction
from Bio.Seq import Seq

seq = Seq("ATGCGCGCAT")
gc = gc_fraction(seq)
print(gc)               # 0.6
print(f"GC 含量: {gc:.1%}")   # GC 含量: 60.0%
```

> 注意： **API 变迁提醒**：早期版本用 `Bio.SeqUtils.GC()`（大写函数），新版本推荐 `gc_fraction()`。旧函数仍可用但已过时，新代码请用 `gc_fraction`。
>
> 提示： `gc_fraction` 默认会忽略非 ACGT 字符（如 N），也可以指定只统计 ACGT： `gc_fraction("ATNG", ignore_ambiguous=False)`

5.2 序列的分子量

```python
from Bio.SeqUtils import molecular_weight
from Bio.Seq import Seq

print(molecular_weight(Seq("ATGC")))   # 双链 DNA 的分子量（默认）
print(molecular_weight(Seq("ATGC"), double_stranded=False))  # 单链
```

5.3 蛋白质理化性质分析 ProtParam

对蛋白质序列，`Bio.SeqUtils.ProtParam.ProteinAnalysis` 能一次性算出一大堆指标：氨基酸组成、分子量、等电点、疏水性、脂肪族氨基酸指数等。这是做蛋白质功能预测、设计实验时的高频工具：

```python
from Bio.SeqUtils.ProtParam import ProteinAnalysis

protein = "MAIVMGRKGAR"   # 一段蛋白质序列

pa = ProteinAnalysis(protein)

# 1. 氨基酸组成（每种氨基酸的数量）
aa_counts = pa.count_amino_acids()
print(aa_counts)          # {'A': 2, 'R': 2, 'G': 2, ...}

# 2. 氨基酸百分含量
aa_percent = pa.get_amino_acids_percent()
print(f"A 占 {aa_percent['A']:.1%}")

# 3. 分子量（道尔顿）
print(f"分子量: {pa.molecular_weight():.1f} Da")

# 4. 等电点 pI（蛋白质带净电荷为零时的 pH）
print(f"等电点: {pa.isoelectric_point():.2f}")

# 5. 脂肪族氨基酸指数（衡量蛋白热稳定性）
print(f"脂肪族指数: {pa.aromaticity():.3f}")

# 6. 总疏水性（GRAVY，负值偏亲水）
print(f"GRAVY: {pa.gravy():.3f}")
```

> 提示： **GRAVY 怎么读**：GRAVY > 0 表示蛋白偏疏水（常与跨膜、膜蛋白相关），GRAVY < 0 偏亲水（常为可溶性蛋白）。它是蛋白质组学入门最常用的指标之一。

5.4 在序列中查找子序列

`Bio.SeqUtils.nt_search` 返回所有匹配位置（第 0 项是匹配模式本身）：

```python
from Bio.SeqUtils import nt_search

positions = nt_search("ATGCATGCAT", "AT")
print(positions)   # ['AT', 0, 4, 8]  —— 在位置 0、4、8 处出现
```

5.5 综合示例：批量计算 GC 含量并统计

把第 4 章与本章结合，写一个真正有用的小工具——统计 FASTA 文件里每条序列的 GC 含量分布：

```python
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

gc_values = []
for record in SeqIO.parse("genes.fasta", "fasta"):
    gc = gc_fraction(record.seq)
    gc_values.append((record.id, gc))
    print(f"{record.id:12s}  GC = {gc:.1%}")

# 找出 GC 含量最高的序列
best = max(gc_values, key=lambda x: x[1])
print(f"GC 最高的是 {best[0]}，含量 {best[1]:.1%}")
```

5.6 小结与练习

本章核心函数：

函数	功能	所在模块
gc_fraction(seq)	GC 含量	Bio.SeqUtils
molecular_weight(seq)	分子量	Bio.SeqUtils
ProteinAnalysis(seq)	蛋白多项性质	Bio.SeqUtils.ProtParam
nt_search(seq, motif)	子序列定位	Bio.SeqUtils

练习：

1. 计算 `"ACGTACGTACGT"` 的 GC 含量（应为 50%）。

2. 用 `ProteinAnalysis` 计算蛋白质 `"MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF"`（人血红蛋白 α 亚基的一段）的分子量与等电点。
3. 写一个函数 `gc_of_file(path)`：读取 FASTA 文件，返回每条记录 ID 到 GC 含量的字典。

第 6 章 序列比对

序列比对的本质是回答一个问题：**两条序列有多像？** 通过将序列排成行、在必要位置插入间隙（gap，用 `-` 表示），让相同或相似的字符对齐，再打分。这是进化分析、功能注释、序列检索的底层技术。

6.1 一个直观的例子

```
序列 A:  ACG-T
序列 B:  ACGGT
         |||-|
```

A 和 B 在 5 个位置上有 4 处匹配，第 4 位插入了一个间隙。这样的排列就是一次"比对"。Biopython 的 `PairwiseAligner` 负责自动寻找最优比对。

6.2 PairwiseAligner：双序列比对

`Bio.Align.PairwiseAligner` 是现代 Biopython（1.80+）推荐的比对工具，API 清晰、性能好：

```python
from Bio.Align import PairwiseAligner

aligner = PairwiseAligner()
aligner.mode = "global"        # 全局比对：从头到尾比对整条序列

seq1 = "ACGT"
seq2 = "ACGGT"

alignments = aligner.align(seq1, seq2)
best = alignments[0]

print(best.score)      # 3.0 —— 比对得分
print(best)            # 打印比对结果
```

输出：

```
target            0 ACG-T 4
                  0 |||-| 5
query             0 ACGGT 5
```

`target` 是第一条序列，`query` 是第二条；中间的 `|` 表示匹配，`.` 表示相似但不相同（蛋白质比对时常见），`-` 是间隙。

6.3 局部比对与参数调节

全局比对 vs 局部比对：全局比对强制两条序列从头到尾对齐，适合相似度高的序列；局部比对（mode="local"）只找最相似的片段，适合找保守结构域、处理长短差异大的序列：

```python
aligner = PairwiseAligner()
aligner.mode = "local"          # 局部比对

seq1 = "GGGGACGTACGTGGGG"
seq2 = "AAAACGTACGTAAAA"
for aln in aligner.align(seq1, seq2):
    print(aln.score)
    print(aln)
```

调节打分参数：比对质量取决于打分规则——匹配得分、错配罚分、间隙开启/延伸罚分。默认参数适合大多数情况，但你可以按需调整：

```python
aligner = PairwiseAligner()
aligner.mode = "global"
aligner.match_score = 2         # 匹配 +2
aligner.mismatch_score = -1     # 错配 -1
aligner.open_gap_score = -2     # 开启一个间隙罚 2 分
aligner.extend_gap_score = -0.5 # 间隙每延伸一个碱基罚 0.5 分

# 蛋白质比对可以指定打分矩阵（如 BLOSUM62）
from Bio.Align import substitution_matrices
blosum62 = substitution_matrices.load("BLOSUM62")
aligner.substitution_matrix = blosum62
```

> 提示：**什么情况调参？** 序列相似度高、长度接近 → 用默认参数即可；要追求灵敏度（找远缘同源）→ 降低 gap 罚分；要追求严谨（防假阳性）→ 提高匹配分数或降低 gap 罚分。

6.4 多序列比对对象 MultipleSeqAlignment

三条及以上序列的比对叫多序列比对（MSA）。Biopython 用 `MultipleSeqAlignment` 表示 MSA 结果。注意：**Biopython 本身不提供 MSA 算法**，它负责读取/写出 MSA 文件（由 MUSCLE、ClustalW、MAFFT 等外部软件生成）：

```python
from Bio.Align import MultipleSeqAlignment
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

# 手工构造一个多序列比对（注意：所有序列长度必须相等）
aln = MultipleSeqAlignment([
    SeqRecord(Seq("ACGTACGT"), id="seqA"),
    SeqRecord(Seq("ACGTACGA"), id="seqB"),
    SeqRecord(Seq("ACGTTCGT"), id="seqC"),
])

print(aln)
# 输出：
# seqA  ACGTACGT
# seqB  ACGTACGA
# seqC  ACGTTCGT

print(aln.get_alignment_length())   # 8 —— 比对后的长度
```

6.5 AlignIO：读写比对文件

MSA 结果通常保存为 Clustal、Stockholm 等格式。`Bio.AlignIO` 与 `SeqIO` 用法几乎一模一样：

```python
from Bio import AlignIO

# 读取比对文件
aln = AlignIO.read("alignment.aln", "clustal")
print(aln.get_alignment_length())

# 写入 Stockholm 格式
AlignIO.write(aln, "alignment.sth", "stockholm")

# 遍历比对中的每条序列
for record in aln:
    print(record.id, record.seq[:20])
```

> 提示： 常见 MSA 工具：**MUSCLE**（快）、**MAFFT**（准）、**Clustal Omega**（经典）。它们都是命令行软件，输出 Clustal/FASTA 格式后，用 AlignIO 读进 Biopython 继续分析。

6.6 实战场景：从比对中找保守位点

读入一个比对文件，统计每个位置的一致性（所有序列在该位置碱基相同即为保守位点）：

```python
from Bio import AlignIO

aln = AlignIO.read("alignment.aln", "clustal")
n = len(aln)                # 序列条数
length = aln.get_alignment_length()

conserved = []
for i in range(length):
    column = aln[:, i]      # 取第 i 列的字符
    if len(set(column)) == 1:   # 该列只有一个字符（全一致）
        conserved.append(i + 1)  # 位置从 1 开始计数

print(f"共 {length} 列，其中保守位点 {len(conserved)} 个：{conserved}")
```

> 注意： 注意 `aln[:, i]` 的切片语法：第一个维度是"序列"，第二个是"列"。取一列返回的是字符串，如 `"ACGT"`。多序列比对对象像一个二维表格，这是它和普通序列最大的区别。

6.7 小结与练习

本章核心 API：

功能	代码
双序列比对	`PairwiseAligner().align(s1, s2)`
全局/局部	`aligner.mode = "global"/"local"`
打分矩阵	`substitution_matrices.load("BLOSUM62")`
多序列比对对象	`MultipleSeqAlignment([...])`
读写比对文件	`AlignIO.read/write`

练习：

1. 全局比对 `"ACGT"` 与 `"ACGGT"`，把得分和比对结果打印出来，对照 6.2 的输出。
2. 把 `aligner.mode` 改成 `"local"`，重新比对 `"GGGGACGTACGTGGGG"` 与 `"AAAACGTACGTAAAA"`，观察局部比对找到的片段。
3. 读入你生成的任意多序列比对，统计非空位列中保守位点的比例。

第 7 章 在线数据库与 BLAST

前面几章我们处理的都是本地文件。但生物信息学的大部分"原材料"来自公共数据库——NCB（基因/蛋白）、UniProt（蛋白功能）、PDB（结构）等。Biopython 提供了访问这些数据库的接口，其中最重要的是 NCBI 的 **Entrez** 系统和 **BLAST** 比对服务。

> 注意： **本章需要联网**。NCBI 对程序化访问有明确规范：**每秒最多 3 次请求、必须携带联系邮箱**（`Entrez.email`）。请遵守规范，做一个文明的爬虫。

7.1 设置邮箱与获取 NCBI 密钥

```python
from Bio import Entrez

Entrez.email = "your_email@example.com"   # 必须设置，NCBI 靠它联系违规用户
# Entrez.api_key = "你的NCBI密钥"         # 可选，注册后获得，可提高请求限额
```

7.2 搜索数据库：Entrez.esearch

在 NCBI 上"检索"就像用搜索引擎，检索结果返回的是命中记录的 ID 列表：

```python
from Bio import Entrez

handle = Entrez.esearch(
    db="nucleotide",          # 数据库：nucleotide/protein/pubmed 等
    term="BRCA1 AND Homo sapiens[Organism]",  # 检索词（支持布尔逻辑与字段限定）
    retmax=10                 # 最多返回多少条 ID
)
result = Entrez.read(handle)
handle.close()

print(result["Count"])        # 命中总数
print(result["IdList"])       # 命中的记录 ID 列表
```

> 提示：**检索词语法**（与 NCBI 网页搜索一致）：支持 `AND`、`OR`、`NOT`；字段限定用 `[字段]`，例如 `Homo sapiens[Organism]`、`complete[Title]`。在 NCBI 网页上试好检索词，再搬到代码里，是最稳的工作流。

7.3 下载序列：Entrez.efetch

拿到 ID 后，用 `efetch` 下载完整记录，配合 `SeqIO` 直接解析：

```python
from Bio import Entrez, SeqIO

# 下载一条序列（比如上面的第一个 ID）
handle = Entrez.efetch(
    db="nucleotide",
    id=result["IdList"][0],   # 换成你搜索到的 ID
    rettype="fasta",          # 返回格式：fasta / genbank
    retmode="text"
)
record = SeqIO.read(handle, "fasta")
handle.close()

print(record.id, len(record.seq))
print(record.seq[:60])
```

> 注意：**为什么要 close()？** `Entrez.read` 和 `SeqIO` 解析后，句柄（handle）还占着网络连接。随手 `close()` 是好习惯，也能避免连接泄漏。

7.4 BLAST 比对：找相似序列

BLAST（Basic Local Alignment Search Tool）是生物信息学最著名的"搜索引擎"——给定一条序列，它在整个数据库里找相似的序列。Biopython 提供两种方式：**远程 BLAST**（调 NCBI 服务器，无需本地安装，但排队较慢）和**本地 BLAST**（需安装 `blast+` 命令行工具，快但需自己建库）。

```python
from Bio.Blast import NCBIWWW, NCBIXML
from Bio import SeqIO

# 读取一条序列（示例用本地 FASTA 文件）
record = SeqIO.read("query.fasta", "fasta")

# 远程 BLAST（NCBI 服务器），可能需要几分钟
result_handle = NCBIWWW.qblast(
    "blastn",           # 程序类型：blastn 核酸比对 / blastp 蛋白比对
    "nt",               # 数据库：nt 核酸库 / nr 蛋白库
    record.seq,         # 查询序列
    hitlist_size=5      # 返回前 5 条结果
)

# 解析 BLAST 结果
blast_records = NCBIXML.parse(result_handle)
for blast_record in blast_records:
    for alignment in blast_record.alignments[:5]:
        for hsp in alignment.hsps:
            print("命中:", alignment.title)
            print(f"   相似度: {hsp.identities}/{hsp.align_length}, "
                  f"E 值: {hsp.expect}")
```

> 提示： **E 值（E-value）怎么读**：E 值代表"这条命中纯属随机巧合的概率"。E 值越小越好——`1e-50` 意味着几乎不可能随机出现，是强同源证据；`0.1` 以上基本不可信。
>
> 注意： 远程 BLAST 通常需要排队 30 秒到几分钟，这是正常现象。生产环境建议使用本地 BLAST：先 `pip install blast` 或用 conda 安装 NCBI BLAST+，再用 `Bio.Blast.Applications.NcbiblastnCommandline` 调用。

7.5 PubMed 文献检索

Entrez 不止能查序列，还能查文献：

```python
from Bio import Entrez

handle = Entrez.esearch(db="pubmed", term="biopython[Title]", retmax=20)
result = Entrez.read(handle)
handle.close()
print(result["IdList"])   # PubMed 文献 ID 列表
```

7.6 小结与练习

本章核心 API：

函数	功能
`Entrez.esearch(db, term)`	搜索数据库，返回 ID 列表
`Entrez.efetch(db, id)`	下载完整记录
`Entrez.read(handle)`	解析 XML 结果
`NCBIWWW.qblast(prog, db, seq)`	远程 BLAST 比对
`NCBIXML.parse(handle)`	解析 BLAST 结果

练习：

1. 在 nucleotide 数据库搜索 `"insulin AND Homo sapiens[Organism]"`，取第一条 ID 下载 FASTA 并打印前 50 个碱基。
2. 把 7.4 的查询序列换成你自己的序列，运行远程 BLAST，输出前 3 条命中的标题与 E 值。
3. 用 `Entrez.esearch` 在 pubmed 搜索 `"BRCA1"`，统计命中数量（`Count`）。

第 8 章 蛋白质三维结构 Bio.PDB

蛋白质的三**维结构**决定其功能。PDB（Protein Data Bank）数据库存储了实验解析的蛋白结构。`Bio.PDB` 模块能读取、分析 PDB 文件，是结构生物学的入门利器。

> 提示：本章涉及结构生物学概念（残基、原子坐标、二级结构）。如果只是想处理序列，可以先跳过本章，用到时再回来看。

8.1 PDB 文件长什么样

PDB 文件是文本文件，核心内容是**每个原子的坐标**。一行典型记录：

```
ATOM      1  N   ALA A   1      11.104   6.134  -6.504  1.00 19.91           N
```

这行表示：1 号原子（N 原子），属于 A 链的 1 号残基 ALA（丙氨酸），三维坐标 (11.104, 6.134, -6.504)。

8.2 用 PDBParser 读取结构

```python
from Bio.PDB import PDBParser

parser = PDBParser(QUIET=True)          # QUIET=True 不打印警告
structure = parser.get_structure("demo", "protein.pdb")

# 结构层级: Structure -> Model -> Chain -> Residue -> Atom
for model in structure:
    print("Model:", model.id)
    for chain in model:
        print("  Chain:", chain.id, "共", len(chain), "个残基")
        for residue in chain:
            print("    Residue:", residue.resname, residue.id[1])
            for atom in residue:
                print("      Atom:", atom.name, atom.coord)
```

> 提示：**结构层级记忆口诀**：S → M → C → R → A（结构 → 模型 → 链 → 残基 → 原子），层层嵌套，像俄罗斯套娃。

8.3 常用操作

遍历所有原子并统计：

```python
from Bio.PDB import PDBParser

structure = PDBParser(QUIET=True).get_structure("p", "protein.pdb")

# 统计原子类型
atom_counts = {}
for atom in structure.get_atoms():
    atom_counts[atom.element] = atom_counts.get(atom.element, 0) + 1
print(atom_counts)   # 如 {'N': 100, 'C': 200, 'O': 150, 'S': 5}

# 获取所有残基名称列表
residues = [res.resname for res in structure.get_residues()]
print(residues[:10])
```

计算两个原子间的距离：

```python
from Bio.PDB import PDBParser
import numpy as np

structure = PDBParser(QUIET=True).get_structure("p", "protein.pdb")
atoms = list(structure.get_atoms())

if len(atoms) >= 2:
    a1, a2 = atoms[0], atoms[1]
    distance = np.linalg.norm(a1.coord - a2.coord)
    print(f"原子 {a1.name} 与 {a2.name} 的距离: {distance:.2f} Å")
```

8.4 蛋白结构可视化

`Bio.PDB` 本身不带渲染功能，但可以输出数据给可视化工具。最流行的选择是 **PyMOL**（独立软件）或 **NGLView**（Jupyter 插件）。在 Jupyter 里配合 matplotlib 也可以简单展示：

```python
# 提取 Cα 原子坐标并绘制主链走向
from Bio.PDB import PDBParser
import matplotlib.pyplot as plt

structure = PDBParser(QUIET=True).get_structure("p", "protein.pdb")
model = structure[0]

xs, ys, zs = [], [], []
for chain in model:
    for residue in chain:
        if "CA" in residue:
            xs.append(residue["CA"].coord[0])
            ys.append(residue["CA"].coord[1])
            zs.append(residue["CA"].coord[2])

plt.figure(figsize=(8, 6))
plt.plot(xs, ys, "-o", markersize=2)
plt.xlabel("X 坐标 (Å)")
plt.ylabel("Y 坐标 (Å)")
plt.title("蛋白质主链 Cα 原子走向")
plt.show()
```

> 注意： PDB 文件可能需要**同源建模或结构预测**的软件配合。2021 年 AlphaFold2 问世后， `Bio.PDB` 配合 AlphaFold 预测结构做分析，已成为结构生信的主流工作流。

8.5 小结与练习

本章核心 API：

功能	代码
读取 PDB	`PDBParser(QUIET=True).get_structure(id, 文件)`
遍历层级	`structure → model → chain → residue → atom`
原子坐标	`atom.coord`（numpy 数组）
残基名称	`residue.resname`
距离计算	`np.linalg.norm(a1.coord - a2.coord)`

练习：

1. 下载任意 PDB 文件（如 `https://files.rcsb.org/download/1CRN.pdb`），统计其中的原子总数和氨基酸残基总数。
2. 找出结构中距离最近的一对原子。
3. 用 matplotlib 画一条链的主链 Cα 轨迹。

第 9 章 系统发育分析 Bio.Phylo

系统发育树（进化树）展示生物或基因之间的**进化关系**——像一张"家族谱系图"。`Bio.Phylo` 负责树的读取、操作和绘制。

9.1 树文件格式：Newick

Newick 是树的标准文本格式，用括号嵌套表示分支。例如：

```
(A:0.1,B:0.2,(C:0.3,D:0.4):0.5);
```

`:` 后是分支长度（进化距离），逗号分隔并列分支，括号嵌套表示共同祖先。

9.2 读取与遍历树

```python
from Bio import Phylo

tree = Phylo.read("tree.nwk", "newick")

# 打印树的结构（文本形式）
Phylo.draw_ascii(tree)

# 遍历所有分支点（Clade）
for clade in tree.find_clades():
    print(clade.name, clade.branch_length)

# 获取所有叶子（终端节点，即实际物种/序列）
leaves = tree.get_terminals()
print([leaf.name for leaf in leaves])
```

9.3 从多序列比对构建进化树

这是最常见的实战流程：**比对** → **计算距离** → **建树**。用第 6 章的 MSA 对象直接开工：

```python
from Bio import Phylo
from Bio.Align import MultipleSeqAlignment
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Phylo.TreeConstruction import DistanceCalculator, DistanceTreeConstructor

# 1. 构造多序列比对（实际工作中从文件读入）
aln = MultipleSeqAlignment([
    SeqRecord(Seq("ACGTACGT"), id="species_A"),
    SeqRecord(Seq("ACGTACGA"), id="species_B"),
    SeqRecord(Seq("ACGTTCGT"), id="species_C"),
    SeqRecord(Seq("ACGTTTGT"), id="species_D"),
])

# 2. 计算两两距离矩阵（"identity" 模型：按一致位点比例）
calculator = DistanceCalculator("identity")
distance_matrix = calculator.get_distance(aln)
print(distance_matrix)

# 3. 用邻接法（Neighbor-Joining）构建进化树
constructor = DistanceTreeConstructor()
tree = constructor.nj(distance_matrix)

# 4. 查看结果
Phylo.draw_ascii(tree)
```

输出示例：

```
          ____________________________________________ species_A
         |
|________|____________________________________________ species_B
         |
         |____________________________________________ species_C
         |
         |____________________________________________ species_D
```

> 提示： 其他建树方法： `constructor.upgma(distance_matrix)` 是 UPGMA 法（假设分子钟）；更常用的最大似然法（如 RAxML、IQ-TREE）需要外部软件，Biopython 负责读入其结果。邻接法 NJ 是"又快又稳"的入门选择。

9.4 美化绘制进化树

用 matplotlib 画出可保存的高质量进化树：

```python
import matplotlib.pyplot as plt
from Bio import Phylo

tree = Phylo.read("tree.nwk", "newick")

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
Phylo.draw(tree, axes=ax, branch_labels=lambda c: c.branch_length)
plt.savefig("tree.png", dpi=300)
plt.show()
```

9.5 树的常用操作

```python
from Bio import Phylo

tree = Phylo.read("tree.nwk", "newick")

# 树的深度（从根到最远叶子的分支长度和）
print(tree.depth())

# 两叶子之间的路径距离
leaf1, leaf2 = tree.get_terminals()[:2]
print(tree.distance(leaf1, leaf2))

# 剪枝：移除某个分支
tree.prune(tree.find_any("some_taxon"))
```

9.6 小结与练习

本章核心 API：

功能	代码
读取树	`Phylo.read(文件, "newick")`
文本绘制	`Phylo.draw_ascii(tree)`
图形绘制	`Phylo.draw(tree, axes=ax)`
距离矩阵	`DistanceCalculator("identity").get_distance(aln)`
NJ 建树	`DistanceTreeConstructor().nj(dm)`
遍历分支	`tree.find_clades()`

练习：

1. 用 9.3 的数据构建 NJ 树，解释为什么 species_A 和 species_B 关系最近（提示：对比它们的序列）。
2. 把自己构造的树保存为 Newick 文件（`Phylo.write(tree, "my.tree", "newick")`），再用 `Phylo.read` 读回来验证。
3. 查阅 `Phylo.draw` 文档，尝试把分支长度显示在树枝上。

第 10 章 其他实用模块

Biopython 是一个巨大的工具箱，除了前面几章的主角，还有一批"小而美"的模块值得了解。知道它们的存在，比会背它们的 API 更重要——需要时知道"有这个功能"就能省下半天功夫。

10.1 限制性内切酶分析 Bio.Restriction

限制性内切酶是分子克隆的核心工具，能识别特定 DNA 序列并切割。`Bio.Restriction` 内置了数千种酶的识别位点信息：

```python
from Bio.Restriction import EcoRI, BamHI, RestrictionBatch
from Bio.Seq import Seq

# 查看酶的识别位点
print(EcoRI.site)              # GAATTC
print(EcoRI.elucidate())       # G^AATT_C —— ^ 表示切点

# 在序列中查找酶切位点
seq = Seq("GAATTCGGATCCGAATTC")
print(EcoRI.search(seq))       # [0, 10] —— 位置 0 和 10 处有 EcoRI 位点

# 同时分析多种酶（分子克隆选酶神器）
enzymes = RestrictionBatch([EcoRI, BamHI])
analysis = enzymes.search(seq)
for enz, sites in analysis.items():
    print(enz, "切割位点:", sites)
```

> 提示： 做分子克隆设计时，可以用它检查目标序列中有没有"不该出现"的酶切位点，避免克隆方案失败。

10.2 序列模体 Bio.motifs

模体（motif）是序列中反复出现的保守短片段，常与功能相关（如转录因子结合位点）。`Bio.motifs` 支持读取 JASPAR、MEME 等格式，也可以从序列集合构造位置权重矩阵：

```python
from Bio import motifs
from Bio.Seq import Seq

# 从一组实例序列构造模体
instances = [Seq("TATAA"), Seq("TATAA"), Seq("TATTA")]
motif = motifs.create(instances)

print(motif.consensus)      # 一致性序列：TATAA
print(motif.counts)         # 每个位置各碱基的出现次数
# 输出：A: [0 0 3 3 0] ...
```

10.3 密码子使用偏好

不同物种对同义密码子的使用有偏好（codon usage bias）。Biopython 提供了密码子使用表模块，可统计序列中的密码子频率：

```python
from Bio.Seq import Seq
from Bio.SeqUtils import CodonUsage

# 统计一条 CDS 序列的密码子使用
seq = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
codon_usage = CodonUsage.CodonAdaptationIndex()
codon_usage.set_cai_index(seq)
print(codon_usage.cai)   # 密码子适应指数（CAI），衡量表达水平

# 也可以直接数密码子
from collections import Counter
codons = [str(seq[i:i+3]) for i in range(0, len(seq) - 2, 3)]
print(Counter(codons))
```

> 提示：密码子优化（codon optimization）是蛋白表达实验的常见需求：把基因序列改成宿主偏好密码子，可显著提高表达量。`CodonUsage` 是入门级工具，生产环境常配合专门的优化软件。

10.4 其他值得一提的模块

模块	功能
`Bio.Graphics`	基因组图形绘制（如 GC 含量曲线图）
`Bio.KEGG`	访问 KEGG 通路数据库
`Bio.PopGen`	群体遗传学分析
`Bio.PDB.MMCIFParser`	读取 mmCIF 格式结构文件（PDB 的新标准格式）
`Bio.SeqFeature`	序列特征（基因、CDS 等）的数据结构

> 提示： **探索技巧**：在 Python 里输入 `import Bio` 后用 `dir(Bio)` 查看所有子模块；或者访问 docs.biopython.org，右上角搜索关键词——官方文档的搜索功能远比想象中好用。

第 11 章 综合实战：三个完整的项目

前面十章是"零件"，这一章把它们组装成"机器"。三个项目由浅入深，建议独立完成前先自己动手写，再对照参考实现。

11.1 项目一：基因序列分析报告生成器

需求：给定一个 FASTA 文件，对每条序列输出 ID、长度、GC 含量、翻译后的蛋白长度，并把报告写入 CSV 文件。

```python
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction
import csv

def analyze_genes(fasta_file, report_file):
    """分析 FASTA 中每条序列，生成 CSV 报告"""
    rows = []
    for record in SeqIO.parse(fasta_file, "fasta"):
        seq = record.seq
        protein_len = len(seq.translate(to_stop=True)) if len(seq) % 3 == 0 else "非整密码子"
        rows.append({
            "id": record.id,
            "length": len(seq),
            "gc_content": round(gc_fraction(seq), 4),
            "protein_len": protein_len,
        })

    with open(report_file, "w", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=["id", "length", "gc_content", "protein_len"])
        writer.writeheader()
        writer.writerows(rows)
    print(f"分析完成，共 {len(rows)} 条序列，报告已保存到 {report_file}")

analyze_genes("genes.fasta", "report.csv")
```

> 提示：**设计要点**：函数化（可复用）、CSV 用 `utf-8` 编码（避免 Excel 打开中文乱码）、异常序列（长度非 3 的倍数）有明确标注。

11.2 项目二：序列清洗与格式整理工具

需求：NCBI 下载的序列常混有不确定碱基（N）、小写字母、或非标准字符。写一个工具做清洗：统一大写、过滤含 N 过多的序列、按长度排序后输出。

```python
from Bio import SeqIO

def clean_fasta(input_file, output_file, max_n_ratio=0.05, min_length=50):
    """清洗序列：统一大写、过滤 N 比例过高和过短的序列"""
    cleaned = []
    for record in SeqIO.parse(input_file, "fasta"):
        seq = str(record.seq).upper()          # 统一大写

        # 过滤 N 比例过高的序列
        n_count = seq.count("N")
        if n_count / len(seq) > max_n_ratio:
            continue

        # 过滤长度不足的序列
        if len(seq) < min_length:
            continue

        record.seq = seq                       # 写回清洗后的序列
        cleaned.append(record)

    # 按长度从长到短排序
    cleaned.sort(key=lambda r: len(r.seq), reverse=True)

    SeqIO.write(cleaned, output_file, "fasta")
    print(f"输入 {input_file} -> 输出 {output_file}，保留 {len(cleaned)} 条")

clean_fasta("raw.fasta", "clean.fasta")
```

> 注意： 注意 `record.seq = seq` 这里 `seq` 必须是 `Seq` 对象或字符串。清洗数据后保留原注释（id、description）是很重要的习惯——数据溯源是生物信息学的职业素养。

11.3 项目三：ORF 查找器

需求：在一条 DNA 序列的 6 个读框（3 个正向 + 3 个反向）中查找所有开放阅读框（ORF，即从起始密码子 ATG 到终止密码子之间的编码区），输出位置和翻译产物。

Python

```python
from Bio.Seq import Seq

def find_orfs(dna, min_length=30):
    """在 DNA 序列的 6 个读框中查找 ORF"""
    seq = Seq(dna.upper())
    results = []

    for strand, template in [(1, seq), (-1, seq.reverse_complement())]:
        for frame in range(3):
            # 逐帧翻译，用 * 分隔 ORF
            length = 3 * ((len(template) - frame) // 3)
            translated = str(template[frame:frame + length].translate())

            # 按终止密码子切分
            pos_in_frame = frame
            for segment in translated.split("*"):
                if len(segment) >= min_length // 3:   # 蛋白长度 >= 10 aa
                    start = pos_in_frame
                    end = pos_in_frame + len(segment) * 3
                    results.append({
                        "strand": strand,
                        "frame": frame + 1,
                        "start": start + 1,          # 转成 1-based 坐标
                        "end": end,
                        "protein": segment,
                    })
                pos_in_frame += (len(segment) + 1) * 3

    return results

dna = "ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAGCCATGGCCATTGTA"
for orf in find_orfs(dna):
    print(f"链 {orf['strand']:+d} 读框 {orf['frame']}  "
          f"位置 {orf['start']}-{orf['end']}  "
          f"蛋白: {orf['protein']}")
```

> 提示：这是"读懂基因"的基础工具——基因注释软件（如 Prodigal）的核心逻辑就是这个思路的工程化扩展。做生物信息学，动手实现一个 ORF 查找器能极大加深对"读框"概念的理解。

11.4 项目四：批量蛋白理化性质分析

需求：读取蛋白质 FASTA 文件，为每条蛋白计算分子量、等电点、GRAVY 值，输出汇总表格，并标记"可能跨膜"（GRAVY > 0.4）的蛋白。

```python
from Bio import SeqIO
from Bio.SeqUtils.ProtParam import ProteinAnalysis

def analyze_proteins(fasta_file, output_file="protein_report.tsv"):
    """批量计算蛋白理化性质"""
    with open(output_file, "w", encoding="utf-8") as out:
        out.write("ID\t长度\t分子量\t等电点\tGRAVY\t可能跨膜\n")
        for record in SeqIO.parse(fasta_file, "fasta"):
            pa = ProteinAnalysis(str(record.seq))
            gravy = pa.gravy()
            transmembrane = "是" if gravy > 0.4 else "否"
            out.write(f"{record.id}\t{len(record.seq)}\t"
                      f"{pa.molecular_weight():.1f}\t"
                      f"{pa.isoelectric_point():.2f}\t"
                      f"{gravy:.3f}\t{transmembrane}\n")
    print(f"完成，报告见 {output_file}")

analyze_proteins("proteins.fasta")
```

第 12 章 调试技巧与常见错误

写代码不报错是不可能的，学会读报错、快速定位问题，是生物信息学工程师的核心能力。本章汇总 Biopython 最常见的"坑"。

12.1 高频错误速查表

报错信息	原因	解决方案
`ValueError: more than one record found`	用 `SeqIO.read` 读了含多条记录的文件	改用 `SeqIO.parse`
`AttributeError: 'str' object has no attribute 'seq'`	把字符串当成了 SeqRecord	确认对象类型；用 `record.seq` 前先 `print(type(record))`
`ValueError: Data in row...`	FASTA/GenBank 文件格式损坏	检查文件编码、换行符；用 `SeqIO.parse` 逐个调试
`TypeError: sequence expected`	把 Seq 对象当字符串拼接	用 `str(seq)` 转换
`Bio.Entrez.Parser.ValidationError`	Entrez 参数名拼写错误	对照文档核对参数名
`urllib.error.HTTPError: 429`	请求太快被 NCBI 限流	放慢请求；设置 `Entrez.email`；加 `time.sleep(1)`
`ValueError: Sequences must all be the same length`	MSA 中序列长度不一致	检查比对文件；用 aligner 重新比对后再建 MSA
`FileNotFoundError`	文件路径错误或当前目录不对	用绝对路径；打印 `os.getcwd()` 检查当前目录

12.2 字符串与 Seq 的转换陷阱

`Seq` 和 `str` 在很多时候可以互操作，但**不是所有时候**。最常见的坑：

```python
from Bio.Seq import Seq

seq = Seq("ATGC")

# 完成： 可以：Seq 可以和字符串比较、拼接（部分操作）
print(seq == "ATGC")          # True

# 注意： 注意：字典键不通用
d1 = {"ATGC": 1}
print(d1.get(seq, "未找到"))  # 未找到！str 键和 Seq 键不相等

# 完成： 推荐做法：需要字符串操作时显式转换
text = str(seq)               # 'ATGC'
print(text.upper())           # ATGC

# 完成： 需要 Seq 操作时从字符串构造
new_seq = Seq(text)
```

> 注意： **黄金法则**：存数据用 `Seq`，传给第三方库（如 pandas、matplotlib 的字符串处理）前先 `str()` 转换；用 dict 做键时统一用 `str(seq)`。

12.3 中文与编码问题

```python
# 读取含中文注释的文件时，显式指定编码
with open("genes.fasta", encoding="utf-8") as handle:
    records = list(SeqIO.parse(handle, "fasta"))

# 写入中文 CSV 时用 utf-8-sig（Excel 打开不乱码）
import csv
with open("report.csv", "w", newline="", encoding="utf-8-sig") as f:
    writer = csv.writer(f)
    writer.writerow(["ID", "描述"])
```

> 提示： Windows 下中文文件默认可能是 GBK 编码，报 `UnicodeDecodeError` 时试试 `encoding="gbk"`。

12.4 网络请求失败处理

```python
import time
from Bio import Entrez

Entrez.email = "you@example.com"

def safe_efetch(db, ids, retries=3):
    """带重试的下载，应对网络抖动"""
    for attempt in range(retries):
        try:
            handle = Entrez.efetch(db=db, id=ids, rettype="fasta", retmode="text")
            data = handle.read()
            handle.close()
            return data
        except Exception as e:
            print(f"第 {attempt+1} 次尝试失败: {e}")
            if attempt < retries - 1:
                time.sleep(2 ** attempt)   # 指数退避：等 1s, 2s, 4s...
    raise RuntimeError("多次重试仍然失败")

data = safe_efetch("nucleotide", "NM_000059")
print(data[:200])
```

> 提示：**指数退避（exponential backoff）**：每次失败后等待时间翻倍，是网络编程的标准技巧。不仅对 NCBI，对任何网络 API 都适用。

12.5 调试三件套

```python
# 1. 打印类型 —— 一半的 Bug 是类型不匹配
print(type(record), type(record.seq))

# 2. 打印长度与内容片段 —— 确认数据真的读了进来
print(len(record), record.seq[:30])

# 3. 小样本试跑 —— 先用 2 条记录验证逻辑，再处理全部数据
sample = list(SeqIO.parse("big.fasta", "fasta"))[:2]
for r in sample:
    print(r.id, len(r.seq))
```

> 提示： 建议用 **Jupyter Notebook** 调试：单元格可以反复执行，配合 `print` 和 `type` 检查，能极大加速排错过程。

第 13 章 结语与进阶路线

恭喜你读完了这本书！到这一步，你已经掌握了 Biopython 的核心：序列对象、文件读写、序列分析、比对、数据库检索、结构分析、进化树，以及最重要的——**用这些工具解决实际问题的能力**。

13.1 官方资源

资源	地址	用途
Biopython 官方文档	docs.biopython.org	权威参考，遇到问题先查这里
Biopython 教程（官方）	biopython.org/wiki/Tutorial	系统学习的第一手资料
Biopython Cookbook	官方教程中的实例章节	实战代码宝库
Stack Overflow	stackoverflow.com/questions/tagged/biopython	提问与检索

13.2 下一步学习路线

```
路线 A：测序数据分析
  FASTQ 格式（Biopython 内置）→ 质控 → 比对（bowtie2/STAR）→ 差异表达（DESeq2）

路线 B：基因组学
  Bio.Graphics 绘制基因组图 → 基因注释 → 比较基因组学

路线 C：结构生物学
  Bio.PDB + AlphaFold → 分子动力学（GROMACS）→ 药物设计

路线 D：数据科学方向
  Biopython + pandas 批量分析 → matplotlib/Plotly 可视化 → 机器学习（scikit-learn）
```

13.3 学习建议（来自实践）

1. **动手永远比看有效**：每章练习一定要自己敲一遍；
2. **用真实数据练习**：去 NCBI 下载真实基因/蛋白序列，比示例数据有意思得多；
3. **学会查文档**：`help(Bio.SeqIO.parse)` 能直接看到函数签名和示例；
4. **读源码**：`import Bio; print(Bio.__file__)` 找到源码位置，直接读源码是最快的进阶方式；

5. **保持联系**：生物信息学社区（生物谷论坛、BioStars、生信技能树公众号）有很多中文资料，遇到问题大胆提问。

> **最后送大家一句话：**生物信息学的本质不是"会调用某个库"，而是"理解生物学问题，并懂得用什么工具去回答"。工具会迭代，但思路是通用的。祝你在生物信息学的路上越走越远！

附录 A：常用代码速查表

A.1 序列操作

```python
from Bio.Seq import Seq

seq = Seq("ATGCATGC")
seq.complement()              # 互补
seq.reverse_complement()      # 反向互补
seq.transcribe()              # 转录
seq.translate()               # 翻译
seq.translate(to_stop=True)   # 翻译至终止密码子
seq[2:5]                      # 切片
seq.count("A")                # 碱基计数
```

A.2 文件读写

```python
from Bio import SeqIO

# 读取
for rec in SeqIO.parse("file.fasta", "fasta"): ...
rec = SeqIO.read("single.fasta", "fasta")

# 写出 / 格式转换
SeqIO.write(records, "out.gb", "genbank")

# 比对文件
from Bio import AlignIO
aln = AlignIO.read("aln.clustal", "clustal")
```

A.3 分析工具

```python
from Bio.SeqUtils import gc_fraction, molecular_weight
from Bio.SeqUtils.ProtParam import ProteinAnalysis

gc_fraction(seq)                          # GC 含量
ProteinAnalysis(protein).molecular_weight()   # 蛋白分子量
ProteinAnalysis(protein).isoelectric_point()  # 等电点
```

A.4 数据库与 BLAST

```python
from Bio import Entrez
Entrez.email = "you@example.com"
result = Entrez.read(Entrez.esearch(db="nucleotide", term="BRCA1", retmax=5))
result["IdList"]                          # ID 列表

from Bio.Blast import NCBIWWW, NCBIXML
handle = NCBIWWW.qblast("blastn", "nt", seq)
records = NCBIXML.parse(handle)
```

A.5 进化树

```python
from Bio import Phylo
from Bio.Phylo.TreeConstruction import DistanceCalculator, DistanceTreeConstructor

tree = Phylo.read("tree.nwk", "newick")
Phylo.draw_ascii(tree)
dm = DistanceCalculator("identity").get_distance(aln)
new_tree = DistanceTreeConstructor().nj(dm)
```

A.6 蛋白质结构

```python
from Bio.PDB import PDBParser
structure = PDBParser(QUIET=True).get_structure("p", "protein.pdb")
atoms = structure.get_atoms()             # 所有原子
residues = structure.get_residues()       # 所有残基
```

附录 B：练习参考答案

第 3 章

练习 1

```python
from Bio.Seq import Seq
seq = Seq("AATTCCGG")
print(seq.complement())          # TTAAGGCC
print(seq.reverse_complement())  # CCGGAATT
```

练习 2

```python
from Bio.Seq import Seq
dna = Seq("ATGGTGGCATGA")
protein = dna.translate(to_stop=True)
print(protein)                   # MVA
```

练习 3

```python
from collections import Counter
seq = "ACGTACGTACGT"
print(Counter(seq))              # Counter({'A': 3, 'C': 3, 'G': 3, 'T': 3})
```

第 4 章

练习 1

```python
from Bio import SeqIO
for record in SeqIO.parse("your_file.fasta", "fasta"):
    print(record.id, len(record.seq))
```

练习 2

```python
from Bio import SeqIO
kept = []
for record in SeqIO.parse("all.fasta", "fasta"):
    protein = record.seq.translate()
    if "*" not in protein[:-1]:     # 内部无终止密码子（允许末尾终止符）
        kept.append(record)
SeqIO.write(kept, "no_internal_stop.fasta", "fasta")
```

练习 3

```python
from Bio import SeqIO
seqs1 = [str(r.seq) for r in SeqIO.parse("data.fasta", "fasta")]
SeqIO.write(SeqIO.parse("data.fasta", "fasta"), "data.gb", "genbank")
seqs2 = [str(r.seq) for r in SeqIO.parse("data.gb", "genbank")]
print(seqs1 == seqs2)   # True
```

第 5 章

练习 1

```python
from Bio.SeqUtils import gc_fraction
print(gc_fraction("ACGTACGTACGT"))   # 0.5
```

练习 2

```python
from Bio.SeqUtils.ProtParam import ProteinAnalysis
pa = ProteinAnalysis("MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHF")
print(f"分子量: {pa.molecular_weight():.1f}")   # 约 15126.5
print(f"等电点: {pa.isoelectric_point():.2f}")  # 约 8.70
```

练习 3

```python
from Bio import SeqIO
from Bio.SeqUtils import gc_fraction

def gc_of_file(path):
    return {r.id: gc_fraction(r.seq) for r in SeqIO.parse(path, "fasta")}
```

第 6 章

练习 3

```python
from Bio import AlignIO

aln = AlignIO.read("alignment.aln", "clustal")
n, L = len(aln), aln.get_alignment_length()
conserved = [i for i in range(L) if len(set(aln[:, i])) == 1]
print(f"保守位点比例: {len(conserved)/L:.1%}")
```

第 7 章

练习 1

```python
from Bio import Entrez, SeqIO
Entrez.email = "you@example.com"

handle = Entrez.esearch(db="nucleotide", term="insulin AND Homo sapiens[Organism]", retmax=1)
ids = Entrez.read(handle)["IdList"]
handle.close()

handle = Entrez.efetch(db="nucleotide", id=ids[0], rettype="fasta", retmode="text")
record = SeqIO.read(handle, "fasta")
handle.close()
print(record.seq[:50])
```

第 9 章

练习 1 参考：species_A 与 species_B 只在最后一个碱基不同（T vs A），与其余序列相比差异最小，因此距离最近、在树上聚为一支。

第 11 章

三个项目已在正文给出完整参考实现，请先独立完成再对照。项目三 ORF 查找器的输出示例：

```
链 +1 读框 1  位置 1-21  蛋白: MAIVMGR
链 +1 读框 1  位置 25-42 蛋白: MAIV
...
```