Python IV: Modules, Packages and Biopython

Almost nothing you do in bioinformatics starts from scratch. Reading compressed files, running BLAST, downloading from NCBI, translating a gene with the right genetic code - somebody has already written and tested that code and packaged it up so you can `import` it. Learning what is available, and how to install and use it, saves you from writing (and debugging) hundreds of lines yourself.

What you'll learn

1. What modules and packages are, and the three ways to `import` them
2. A tour of the **standard library** (comes with Python) for data work: `pathlib`/`os`, `sys`, `gzip`, `csv`, `math`, `statistics`, `random`, `collections`, `itertools`, `urllib.request` and `subprocess` (run BLAST from Python)
3. Proper command-line scripts with `argparse`
4. Installing packages: `pip` vs `conda`, virtual environments, `requirements.txt` and `environment.yml`
5. **Biopython**: `Seq`, `SeqRecord`, reading and writing FASTA/FASTQ/GenBank with `SeqIO`, GenBank features, quality scores, and downloading from NCBI with `Entrez`
6. Other domain packages you will meet in the workshops

The next lecture, Pandas, is the other half of Python IV: working with tables.

This builds on Python III, where you wrote your own functions, your own module and a simple FASTA reader. Here you'll see how the same jobs are done with packages.

Data used in this lecture

```
mkdir -p ~/bigdata/python4
cd ~/bigdata/python4
BASE=https://github.com/biodataprog/GEN220_data/raw/main/genome
curl -L -O $BASE/S_cerevisiae.ORFs.fasta.gz        # yeast ORFs (DNA, 6713 genes)
curl -L -O $BASE/S_cerevisiae.fasta.gz             # yeast genome (17 chromosomes)
curl -L -O $BASE/S_cerevisiae.gff3.gz              # yeast genome annotation
curl -L -O $BASE/E_coli_K12.pep.gz                 # E. coli K-12 proteins
curl -L -O $BASE/Ecoli-vs-Senterica.BLASTP.tab.gz  # E. coli vs Salmonella BLASTP
```

`curl -L` follows the redirect that GitHub uses for these files. A few more files are downloaded from NCBI and UniProt by the Python examples themselves.

The examples were run with Python 3.14, Biopython 1.88 and BLAST+ 2.17.0. Anything from Python 3.9 and Biopython 1.80 on should give the same results.

Modules and packages

What is a module?

A **module** is a file of Python code (functions, variables) that you can load into your program. In Python III you made your own - any `myfunctions.py` file in the same folder can be loaded with `import myfunctions`.

A **package** is a folder of modules that belong together, e.g. `Bio` (Biopython) has `Bio.Seq`, `Bio.SeqIO`, `Bio.Entrez` and many more inside it. People often use "module", "package" and "library" to mean the same thing.

Where do they come from?

- **The standard library**: about 200 modules that come with every Python installation (`math`, `gzip`, `csv`, ...). Nothing to install.
- **Third-party packages**: written by other people, installed with `pip` or `conda` (Biopython, pandas, matplotlib, ...). The Python Package Index lists more than half a million.

Three ways to import

```
import math
print(math.log2(8))

from math import log2, sqrt
print(log2(1024), sqrt(16))

import statistics as st
print(st.mean([315, 255, 363]))
```

```
3.0
10.0 4.0
311
```

Form	Then use it as	When to use
`import math`	`math.log2(8)`	the default; it is clear where `log2` came from
`from math import log2`	`log2(8)`	you use a few names very often
`import pandas as pd`	`pd.read_csv(...)`	long names with a standard short nickname (`pd`, `np`, `plt`)

Avoid `from math import *` (import everything): you can't tell where names came from, and one module's names can silently replace another's.

Put all your imports at the top of the script. If an import fails you find out immediately, not halfway through a long run:

```
ModuleNotFoundError: No module named 'Bio'
```

means the package is not installed in the Python you are running (or you have not activated your environment - see Installing packages below). Note that the name you install and the name you import can differ: you `pip install biopython` but `import Bio`.

Finding out what is in a module

```
import math
help(math.log)          # the documentation for one function (q to quit)
print(dir(math)[:8])    # the names defined in the module
```

The real documentation is on the web: https://docs.python.org/3/library/ for the standard library, and each package's own site. Searching "python gzip open" usually finds the right page.

A tour of the standard library for data work

`pathlib` and `os`: files and folders

`pathlib` treats file names as objects that know how to split themselves into parts, check if they exist, and join with folder names. It replaces most of the older `os.path` functions.

```
from pathlib import Path

data = Path("data")                 # a folder name, as a Path object
data.mkdir(exist_ok=True)           # like mkdir -p

gff = Path("S_cerevisiae.gff3.gz")
```

```
print(gff.exists(), gff.name, gff.suffix, gff.suffixes)
print(gff.name.removesuffix(".gff3.gz"))
print(gff.stat().st_size, "bytes")

outfile = data / "yeast_genes.tsv"  # / joins folder and file names
print(outfile)

for f in sorted(Path(".").glob("*.gz")):   # like ls *.gz
    print(f.name)
```

```
True S_cerevisiae.gff3.gz .gz ['.gff3', '.gz']
S_cerevisiae
1152929 bytes
data/yeast_genes.tsv
E_coli_K12.pep.gz
Ecoli-vs-Senterica.BLASTP.tab.gz
S_cerevisiae.ORFs.fasta.gz
S_cerevisiae.fasta.gz
S_cerevisiae.gff3.gz
```

A common pattern is to loop over every input file in a folder and make a matching output name: `for f in Path("reads").glob("*.fastq.gz"): ... out = Path("results") / (f.name.removesuffix(".fastq.gz") + ".stats.tsv")`.

You will also see the older `os` module in many scripts. It does the same jobs:

```
import os

print(os.path.exists("S_cerevisiae.gff3.gz"))
print(os.path.join("data", "yeast_genes.tsv"))
print(os.path.basename("/bigdata/gen220/shared/data/vector.fasta"))
os.makedirs("results/blast", exist_ok=True)
print(os.listdir("results"))
```

```
True
data/yeast_genes.tsv
vector.fasta
['blast']
```

`os.environ` is a dictionary of the environment variables, e.g. `os.environ["HOME"]` is the same as `$HOME` in bash.

sys: talking to the command line

You met `sys.argv` in Python II. `sys` also gives you the standard error stream and a way to stop the program with an error code - just like the exit codes of UNIX commands:

```
import sys

if len(sys.argv) < 2:
    print("usage: count.py FILE", file=sys.stderr)   # errors go to stderr
    sys.exit(1)                                      # non-zero = failure
```

For anything more than one argument, use `argparse` (below).

gzip: read and write compressed files

Sequence data is almost always compressed (`.gz`). `gzip.open()` works just like `open()`, so you don't need to uncompress the file first. Use mode `"rt"` (read text); the default `"rb"` gives you bytes, not strings.

```
import gzip

n_seqs = 0
with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:   # "rt" = read text
    for line in fh:
        if line.startswith(">"):
            n_seqs += 1
print(n_seqs, "ORFs")

# write a compressed file: "wt" = write text
with gzip.open("first_ids.txt.gz", "wt") as out:
    out.write("YAL001C\nYAL002W\n")
```

```
6713 ORFs
```

A small function that opens a file whether or not it is compressed is handy in almost every script (it is used in the `argparse` example below):

```
def open_file(filename):
    """Open a plain or gzip-compressed text file for reading."""
    if filename.endswith(".gz"):
        return gzip.open(filename, "rt")
    return open(filename)
```

`csv`: delimited tables

Python II introduced the `csv` module. With `DictReader` you can give names to the columns of a file that has no header, such as BLAST `-outfmt 6` output (see the BLAST lecture for the 12 columns), and then use the names instead of column numbers:

```
import csv
import gzip

columns = ["qseqid", "sseqid", "pident", "length", "mismatch", "gapopen",
           "qstart", "qend", "sstart", "send", "evalue", "bitscore"]
n_strong = 0
with gzip.open("Ecoli-vs-Senterica.BLASTP.tab.gz", "rt") as fh:
    reader = csv.DictReader(fh, fieldnames=columns, delimiter="\t")
    for row in reader:
        if float(row["pident"]) >= 90 and int(row["length"]) >= 100:
            n_strong += 1
print(n_strong, "hits with >= 90% identity over >= 100 aa")
```

```
1544 hits with >= 90% identity over >= 100 aa
```

Every value comes back as a string, so convert with `float()` / `int()` before comparing. For bigger table jobs (sorting, grouping, joining) use pandas.

`math` and `statistics`

```
import gzip
import math
import statistics

# length of every ORF: add up the sequence lines after each header
lengths = []
with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    for line in fh:
```

```
        if line.startswith(">"):
            lengths.append(0)                   # start a new sequence
        else:
            lengths[-1] += len(line.strip())    # add to the last one

print("n      =", len(lengths))
print("mean   =", round(statistics.mean(lengths), 1))
print("median =", statistics.median(lengths))
print("stdev  =", round(statistics.stdev(lengths), 1))
print("longest =", max(lengths), "bp; log10 =", round(math.log10(max(lengths)), 2))
print("log2 fold change 120 -> 480 reads:", math.log2(480 / 120))
```

```
n      = 6713
mean   = 1352.4
median = 1077
stdev  = 1139.7
longest = 14733 bp; log10 = 4.17
log2 fold change 120 -> 480 reads: 2.0
```

The mean is bigger than the median: gene lengths are skewed, with a long tail of very long genes. `math` also has `log`, `log10`, `exp`, `sqrt`, `ceil`, `floor` and the constants `math.pi` and `math.inf`. For large arrays of numbers you will use `numpy` and `pandas`, which do the same math on whole columns at once.

`random`: simulations and sampling (with seeds)

Random numbers are used to simulate sequences, subsample reads, shuffle sequences to build a null distribution (see Sequence evolution), or pick random genes as a control set.

```
import random

random.seed(220)                        # same seed -> same "random" numbers
dna = "".join(random.choice("ACGT") for i in range(30))
print(dna)

genes = ["ADH1", "ADH2", "ADH3", "ADH4", "ADH5", "SFA1"]
print(random.sample(genes, 3))          # pick 3 without replacement
bases = list(dna)
random.shuffle(bases)                   # shuffle in place, keeps composition
print("".join(bases))
print(random.random(), random.randint(1, 100))
```

```
CCTGTTTCGCTCTGTCAACAGGAGAACACG
['SFA1', 'ADH3', 'ADH1']
AAGCATAGCACTTTCCCTTGGCGTAAGCCG
0.8210222828283064 40
```

Run it twice: you get exactly the same output, because of `random.seed(220)`. Without a seed you get different numbers every time. **Always set a seed in an analysis** so that you (and your reviewers) can reproduce the result, and report it in your methods.

`collections`: `Counter` and `defaultdict`

`Counter` is a dictionary made for counting. `defaultdict` is a dictionary that creates a starting value (e.g. an empty list) the first time you use a new key, so you don't need the `if key not in d:` check from Python III.

```
from collections import Counter, defaultdict

seq = "ATGGCGGCGTTAAGCGGCTAA"
```

```
counts = Counter(seq)
print(counts)
print(counts["G"], counts["N"])       # missing keys count as 0
codons = Counter(seq[i:i+3] for i in range(0, len(seq) - 2, 3))
print(codons.most_common(2))

# group BLAST hits by query: a dictionary whose values start as empty lists
hits = [("geneA", "hit1"), ("geneB", "hit7"), ("geneA", "hit2")]
by_query = defaultdict(list)
for query, subject in hits:
    by_query[query].append(subject)   # no need to check "if query in ..."
print(dict(by_query))
```

```
Counter({'G': 8, 'A': 5, 'T': 4, 'C': 4})
8 0
[('GCG', 2), ('ATG', 1)]
{'geneA': ['hit1', 'hit2'], 'geneB': ['hit7']}
```

`itertools` (briefly)

Tools for looping: all pairs of samples, all possible k-mers, and more.

```
import itertools

samples = ["wt", "mutA", "mutB"]
for a, b in itertools.combinations(samples, 2):
    print(a, "vs", b)

codons = ["".join(c) for c in itertools.product("ACGT", repeat=3)]
print(len(codons), codons[:5])
```

```
wt vs mutA
wt vs mutB
mutA vs mutB
64 ['AAA', 'AAC', 'AAG', 'AAT', 'ACA']
```

`urllib.request`: download files

`urllib.request` does what `curl` does. Check whether the file already exists so you don't download it again every time you run your script.

```
import os
import urllib.request

url = "https://rest.uniprot.org/uniprotkb/P10127.fasta"   # yeast ADH4 protein
outfile = "ADH4.fasta"
if not os.path.exists(outfile):
    urllib.request.urlretrieve(url, outfile)   # like curl -o ADH4.fasta URL

with open(outfile) as fh:
    print(fh.readline()[:60])        # the first 60 characters of the header

# or read the web page straight into memory (it arrives as bytes)
with urllib.request.urlopen(url) as response:
    text = response.read().decode("utf-8")
print(len(text.splitlines()), "lines")
```

```
>sp|P10127|ADH4_YEAST Alcohol dehydrogenase 4 OS=Saccharomyc
8 lines
```

(UniProt's older `www.uniprot.org/uniprot/P10127.fasta` style of URL still redirects to the new `rest.uniprot.org` one.) For NCBI, use Biopython's `Entrez` module (below), which handles NCBI's rules for you. `pandas` can also read a table directly from a URL.

subprocess: run command-line programs from Python

Many bioinformatics tools (BLAST, samtools, bwa ...) are command-line programs. `subprocess.run()` runs one from inside Python, the same as typing it in bash. This lets you glue steps together: prepare an input file in Python, run the tool, and read its output back in.

Let's ask: which *E. coli* proteins are related to yeast ADH4 (the file we just downloaded)? On the cluster, `module load ncbi-blast` first (check `module avail blast` for the current name).

```
import gzip
import os
import shutil
import subprocess

# makeblastdb needs an uncompressed FASTA file
if not os.path.exists("E_coli_K12.pep"):
    with gzip.open("E_coli_K12.pep.gz", "rb") as fin:
        with open("E_coli_K12.pep", "wb") as fout:
            shutil.copyfileobj(fin, fout)     # copy, uncompressing on the way

# each part of the command is a separate string in a list
subprocess.run(["makeblastdb", "-in", "E_coli_K12.pep", "-dbtype", "prot",
                "-out", "E_coli_K12"],
               check=True, capture_output=True)

cmd = ["blastp", "-query", "ADH4.fasta", "-db", "E_coli_K12",
       "-evalue", "1e-10", "-outfmt", "6 sseqid pident length evalue bitscore"]
result = subprocess.run(cmd, check=True, capture_output=True, text=True)

for line in result.stdout.splitlines():
    print(line)
```

```
gi|388479651|ref|YP_491845.1|   55.703  377 1.40e-149   425
gi|388478815|ref|YP_491007.1|   39.362  376 4.12e-89    271
gi|388478488|ref|YP_490680.1|   33.158  380 1.80e-66    213
gi|388477319|ref|YP_489507.1|   30.435  391 1.17e-51    182
gi|388479012|ref|YP_491204.1|   24.202  376 1.35e-20    89.4
```

These are the iron-containing alcohol dehydrogenases of *E. coli* (YqhD, FucO, EutG, the AdhE domain ...) - ADH4 belongs to this family, unlike the other yeast ADHs.

What the options mean:

- **The command is a list**: `["blastp", "-query", "ADH4.fasta", ...]`, one string per word. Python passes them to the program exactly as given, so spaces or odd characters in file names cause no trouble. (`subprocess.run("blastp -query ...", shell=True)` also works but is easy to get wrong; avoid it unless you need pipes.)
- **`check=True`**: if the program fails (exit code not 0), Python stops with a `CalledProcessError`, like `set -e` in bash. Without it your script carries on with missing results.
- **`capture_output=True, text=True`**: collect what the program prints into `result.stdout` and `result.stderr` as strings, instead of printing it to the screen.

- To save a big output to a file, give the program its own output option (`-out hits.tsv`) and then read the file.

For a long pipeline of many tools, a bash script or a SLURM job (UNIX IV) is often simpler; `subprocess` shines when Python has to make decisions between steps.

Command-line scripts with `argparse`

`sys.argv` is fine for one or two arguments, but real tools have options with defaults, checks and a `-h` help message. `argparse` builds all of that for you. Here is a complete script, `seq_lengths.py`:

```
#!/usr/bin/env python3
"""Report the length of each sequence in a FASTA file (plain or .gz)."""
import argparse
import gzip
import sys


def open_file(filename):
    """Open a plain or gzip-compressed text file for reading."""
    if filename.endswith(".gz"):
        return gzip.open(filename, "rt")
    return open(filename)


def read_lengths(filename):
    """Return a dictionary of sequence ID -> length."""
    lengths = {}
    seq_id = None
    with open_file(filename) as fh:
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                seq_id = line[1:].split()[0]      # first word after >
                lengths[seq_id] = 0
            elif seq_id is not None:
                lengths[seq_id] += len(line)
    return lengths


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument("fasta", help="input FASTA file")
    parser.add_argument("-m", "--min-length", type=int, default=0,
                        help="only report sequences at least this long (default: 0)")
    parser.add_argument("-o", "--out", default="-",
                        help="output file (default: - means the screen)")
    parser.add_argument("--summary", action="store_true",
                        help="print only the number and total length")
    args = parser.parse_args()

    lengths = read_lengths(args.fasta)
    keep = {name: n for name, n in lengths.items() if n >= args.min_length}

    if args.summary:
        print(len(keep), "sequences,", sum(keep.values()), "bp")
```

```
        return
    out = sys.stdout if args.out == "-" else open(args.out, "w")
    for name, n in keep.items():
        out.write(f"{name}\t{n}\n")
    if out is not sys.stdout:
        out.close()


if __name__ == "__main__":
    main()
```

You get a help message for free:

```
python3 seq_lengths.py -h
```

```
usage: seq_lengths.py [-h] [-m MIN_LENGTH] [-o OUT] [--summary] fasta

Report the length of each sequence in a FASTA file (plain or .gz).

positional arguments:
  fasta                 input FASTA file

options:
  -h, --help            show this help message and exit
  -m, --min-length MIN_LENGTH
                        only report sequences at least this long (default: 0)
  -o, --out OUT         output file (default: - means the screen)
  --summary             print only the number and total length
```

(Python before 3.13 prints `-m MIN_LENGTH, --min-length MIN_LENGTH`.) Using it:

```
python3 seq_lengths.py S_cerevisiae.fasta.gz | head -3
python3 seq_lengths.py S_cerevisiae.ORFs.fasta.gz --min-length 10000 --summary
python3 seq_lengths.py S_cerevisiae.ORFs.fasta.gz -m 10000 -o long_orfs.tsv
cat long_orfs.tsv
python3 seq_lengths.py S_cerevisiae.ORFs.fasta.gz -m ten
```

```
chrI    230218
chrII   813184
chrIII  316620
3 sequences, 38247 bp
YHR099W 11235
YKR054C 12279
YLR106C 14733
usage: seq_lengths.py [-h] [-m MIN_LENGTH] [-o OUT] [--summary] fasta
seq_lengths.py: error: argument -m/--min-length: invalid int value: 'ten'
```

How it works:

Code	Meaning
`add_argument("fasta")`	a **positional** (required) argument; `args.fasta`
`add_argument("-m", "--min-length", type=int, default=0)`	an **option** with a short and long name; converted to `int`; `args.min_length` (dashes become `_`)
`action="store_true"`	a **flag** with no value: `True` if given, else `False`
`required=True`	make an option mandatory
`choices=["blastn", "blastp"]`	only allow these values
`nargs="+"`	one or more values, e.g. several input files, as a list

`if __name__ == "__main__": main()` runs `main()` only when the file is run as a script, not when another script does `import seq_lengths` to reuse `read_lengths()`. The homework scripts use this layout.

Installing packages

pip, conda and environments

There are two installers you will use:

- **`pip`** installs Python packages from PyPI. It only knows about Python packages.
- **`conda`** (or its faster twin `mamba`) installs packages from channels such as conda-forge and Bioconda. It installs Python packages **and** non-Python programs (BLAST, samtools, R ...), which makes it the usual choice for bioinformatics.

Either way, install into an **environment**: a separate folder with its own Python and its own packages, one per project. This keeps projects from breaking each other when one needs a different version, and lets you delete and rebuild everything if it gets messed up. Never `sudo pip install` (you can't on the cluster anyway), and avoid `pip install --user` - it puts packages where every Python sees them.

On the HPCC cluster: conda

The one-time conda setup on the HPCC (putting environments in `~/bigdata`, the channel settings, installing on a compute node) is in UNIX IV: Software environments. Once that is done, an environment for this lecture is:

```
conda create -n gen220 python=3.12 biopython pandas pyarrow blast
conda activate gen220
python -c "import Bio, pandas; print(Bio.__version__, pandas.__version__)"
```

(Older notes use `source activate`; the current command is `conda activate`.)

To make the environment available as a kernel in Jupyter (e.g. in OnDemand), install `ipykernel` into it and register it once:

```
conda activate gen220
conda install ipykernel
python -m ipykernel install --user --name gen220 --display-name "Python (gen220)"
```

Virtual environments with `venv` and pip

Python's built-in alternative, good on your laptop or when every package you need is on PyPI:

```
python3 -m venv ~/bigdata/venvs/gen220        # create (once)
source ~/bigdata/venvs/gen220/bin/activate    # activate: your prompt shows (gen220)
pip install biopython pandas pyarrow
python -c "import Bio; print(Bio.__version__)"
deactivate                                    # leave the environment
```

Recording what you installed

So that someone else (or you, in six months, on another computer) can rebuild the same environment, save the list of packages alongside your code in git:

```
# pip: write the exact versions installed, then rebuild elsewhere
pip freeze > requirements.txt
pip install -r requirements.txt

# conda: write the packages you asked for, then rebuild elsewhere
conda env export --from-history > environment.yml
conda env create -f environment.yml
```

A hand-written `environment.yml` is short and readable:

```
name: gen220
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.12
  - biopython
  - pandas
  - pyarrow
  - blast
```

Useful checks: `pip list` or `conda list` (what is installed), `which python` (which Python am I running?), `python -c "import Bio; print(Bio.__file__)"` (where did this package come from?).

Biopython

Biopython is the standard Python package for biological sequences and file formats. The Biopython Tutorial and Cookbook is excellent; this section covers the parts you'll use most. Install with `pip install biopython` or `conda install biopython`, and import from `Bio`.

`Seq`: a sequence object

A `Seq` behaves like a string (length, slicing, `count`, `==`) with biology methods added:

```
from Bio.Seq import Seq

dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
print(dna)
print(len(dna), dna.count("G"))
print(dna[0:6])                    # slicing works like a string
print(dna.complement())
print(dna.reverse_complement())
print(dna.transcribe())            # DNA -> mRNA (T -> U)
print(dna.translate())             # standard code; * is a stop codon
print(dna.translate(to_stop=True))
```

```
ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG
39 14
ATGGCC
TACCGGTAACATTACCCGGCGACTTTCCCACGGGCTATC
CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT
AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG
MAIVMGR*KGAR*
MAIVMGR
```

`gc_fraction` (in `Bio.SeqUtils`) gives the GC content, and `str()` turns a `Seq` back into an ordinary string when you need one:

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

dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
print(round(gc_fraction(dna), 3))
print(str(dna).lower())            # str() turns a Seq back into a plain string
print(dna == "ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
```

```
0.564
atggccattgtaatgggccgctgaaagggtgcccgatag
True
```

(Older tutorials use `from Bio.Alphabet import ...` and `GC()`. Alphabets were removed in Biopython 1.78 and `GC()` in 1.82 - if you copy old code that uses them, just delete the alphabet argument and use `gc_fraction`.)

Genetic codes

`translate()` uses the standard code (NCBI table 1) unless you ask for another one. Mitochondria, many bacteria and some nuclear genomes use different codes - the NCBI genetic code tables are numbered, and Biopython knows them all.

```
from Bio.Seq import Seq
from Bio.Data import CodonTable

cds = Seq("ATGTGAATACTTTAA")
print(cds.translate())             # table 1: the standard code
print(cds.translate(table=3))      # table 3: yeast mitochondrial code
print(cds.translate(table="Yeast Mitochondrial"))   # same, by name

yeast_mito = CodonTable.unambiguous_dna_by_id[3]
print(yeast_mito.stop_codons)
print(yeast_mito.forward_table["TGA"], yeast_mito.forward_table["CTT"])
```

```
M*IL*
MWMT*
MWMT*
['TAA', 'TAG']
W T
```

In yeast mitochondria `TGA` is tryptophan, not stop; `ATA` is methionine; and `CTN` is threonine, not leucine. Bacterial genes use table 11 (the same amino acids as table 1, but more start codons). Add `cds=True` when you translate a complete gene: Biopython then checks that it starts with a start codon (and translates it as M even if it is `GTG` or `TTG`), ends with a stop codon, has no stop in the middle and is a multiple of 3, and raises an error if not.

`SeqRecord`: a sequence plus its name and annotation

When you read a file you get `SeqRecord` objects: a `Seq` in `.seq` plus `.id`, `.description`, and for richer formats, `.features`, `.annotations` and `.letter_annotations` (e.g. quality scores). You can also make one yourself:

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

record = SeqRecord(Seq("ATGGCCATTGTAATGGGCCGCTGA"),
                   id="gene1", description="a made-up example gene")
print(record)
print(record.id, len(record))
print(record.format("fasta"))
```

```
ID: gene1
Name: <unknown name>
Description: a made-up example gene
Number of features: 0
Seq('ATGGCCATTGTAATGGGCCGCTGA')
gene1 24
>gene1 a made-up example gene
ATGGCCATTGTAATGGGCCGCTGA
```

`SeqIO.parse`: read sequence files

`SeqIO.parse(file, format)` reads a file one record at a time. The same loop works for `"fasta"`, `"fastq"`, `"genbank"`, `"embl"`, `"clustal"` and many other formats - just change the format name. This replaces the FASTA reader you wrote in Python III.

`SeqIO` can't uncompress by itself, so open `.gz` files with `gzip.open(..., "rt")` and give it the file handle:

```
import gzip
from Bio import SeqIO

with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    for record in SeqIO.parse(fh, "fasta"):
        print(record.id)
        print(record.description[:60])
        print(repr(record.seq))
        print(len(record))
        break           # just look at the first record
```

```
YAL001C
YAL001C TFC3 SGDID:S000000001, Chr I from 151006-147594,1511
Seq('ATGGTACTGACGATTTATCCTGACGAACTCGTACAAATAGTGTCTGATAAAATT...TAA')
3483
```

For FASTA, `.id` is the first word of the header line and `.description` is the whole line (without the `>`).

`SeqIO.write`: filter, translate and save

Make a list of records (or any loop that produces records) and write them in one call. This script keeps the yeast ORFs of at least 6 kb, and also writes their protein translations:

```
#!/usr/bin/env python3
# Write the long yeast ORFs, and their translations, to new FASTA files
import gzip
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord

long_orfs = []
proteins = []
with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    for record in SeqIO.parse(fh, "fasta"):
        if len(record) < 6000:
            continue
        long_orfs.append(record)
        protein = record.seq.translate(to_stop=True)
        proteins.append(SeqRecord(protein, id=record.id, description=""))

n = SeqIO.write(long_orfs, "long_orfs.fasta", "fasta")
print("wrote", n, "ORFs")
n = SeqIO.write(proteins, "long_orfs.pep.fasta", "fasta")
print("wrote", n, "proteins")
print(proteins[0].id, len(proteins[0]), proteins[0].seq[:40])
```

```
wrote 37 ORFs
wrote 37 proteins
YBL004W 2493 MAKQRQTTKSSKRYRYSSFKARIDDLKIEPARNLEKRVHD
```

```
head -3 long_orfs.pep.fasta
```

```
>YBL004W
MAKQRQTTKSSKRYRYSSFKARIDDLKIEPARNLEKRVHDYVESSHFLASFDQWKEINLS
AKFTEFAAEIEHDVQTLPQILYHDKKIFNSLVSFINFHDEFSLQPLLDLLAQFCHDLGPD
```

`SeqIO.write()` returns the number of records written and wraps FASTA sequences at 60 letters per line. Setting `description=""` keeps the header to just the ID.

Looking up sequences by name: `to_dict` and `index`

To pull out particular sequences by their ID, load them into a dictionary with `SeqIO.to_dict()`. This reads everything into memory - fine for a proteome or a set of genes:

```
import gzip
from Bio import SeqIO

with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    orfs = SeqIO.to_dict(SeqIO.parse(fh, "fasta"))

print(len(orfs), "ORFs loaded")
act1 = orfs["YFL039C"]            # look up by ID, like any dictionary
print(act1.description[:40])
print(len(act1), act1.seq[:30])
print("YFL039C" in orfs, "YFL999W" in orfs)
```

```
6713 ORFs loaded
YFL039C ACT1 SGDID:S000001855, Chr VI fr
1128 ATGGATTCTGAGGTTGCTGCTTTGGTTATT
True False
```

For big files (genomes, millions of reads) use `SeqIO.index()` instead. It only records *where* each record starts in the file, and reads a sequence from disk when you ask for it, so it uses very little memory. It needs an uncompressed file (or one compressed with `bgzip`):

```
gunzip -k S_cerevisiae.fasta.gz      # -k keeps the .gz file too
```

```
from Bio import SeqIO

genome = SeqIO.index("S_cerevisiae.fasta", "fasta")   # file must be uncompressed
print(len(genome), "sequences:", list(genome)[:4])

chrVI = genome["chrVI"]            # read from the file only now
print(chrVI.id, len(chrVI))

# ACT1 (YFL039C) is at chrVI:53260-54696 on the - strand (1-based, from the GFF)
act1 = chrVI.seq[53260 - 1:54696].reverse_complement()
print(len(act1), act1[:30])
genome.close()
```

```
17 sequences: ['chrI', 'chrII', 'chrIII', 'chrIV']
chrVI 270161
1437 ATGGATTCTGGTATGTTCTAGCGCTTGCAC
```

Two lessons in this output. First, coordinates: GFF and GenBank files count from 1 and include the end; Python slices count from 0 and exclude the end, so 1-based `start..end` is the slice `[start - 1:end]`. Second, biology: the gene region is 1437 bp but the ORF is 1128 bp - ACT1 has an intron right after `ATGGATTCTG` (introns start with `GT`).

GenBank files and features

GenBank (and EMBL) files carry a sequence plus its **features**: genes, CDS, RNAs, each with a location and **qualifiers** (`/gene=`, `/product=`, `/translation=` ...). Let's download a small, famous genome - bacteriophage phiX174, the control spiked into most Illumina sequencing runs - from NCBI using the `Entrez` module (explained below):

```
#!/usr/bin/env python3
# Download a GenBank record from NCBI and save it to a file
import os
from Bio import Entrez

Entrez.email = "your.name@ucr.edu"     # NCBI requires an email address

accession = "NC_001422"                 # bacteriophage phiX174 genome
outfile = accession + ".gbk"

if not os.path.exists(outfile):         # only download once
    handle = Entrez.efetch(db="nuccore", id=accession,
                           rettype="gbwithparts", retmode="text")
    with open(outfile, "w") as out:
        out.write(handle.read())
    handle.close()
    print("saved", outfile)
else:
    print(outfile, "already downloaded")
```

```
saved NC_001422.gbk
```

(Without Python: `curl -o NC_001422.gbk "https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccor`
The top of the file and one feature look like this:

```
LOCUS       NC_001422               5386 bp ss-DNA     circular PHG 11-JAN-2023
DEFINITION  Escherichia phage phiX174, complete genome.
ACCESSION   NC_001422
VERSION     NC_001422.1
...
FEATURES             Location/Qualifiers
...
     CDS             join(3981..5386,1..136)
                     /locus_tag="phiX174p01"
                     /function="viral strand synthesis"
                     /note="rf replication"
                     /codon_start=1
                     /transl_table=11
                     /product="DNA replication initiation"
                     /protein_id="NP_040703.1"
                     /db_xref="GeneID:2546398"
                     /translation="MVRSYYPSECHADYFDFERIEALKPAIEACGISTLSQSPMLGFH
                     ...
```

Reading it is the same `SeqIO` code with the format `"genbank"`. Here the file has exactly one record, so we use `SeqIO.read()` (it raises an error if there are zero or several records):

```
from Bio import SeqIO

record = SeqIO.read("NC_001422.gbk", "genbank")   # exactly one record
print(record.id, len(record), "bp")
print(record.description)
```

```
print(record.annotations["organism"], record.annotations["topology"])
print(len(record.features), "features")
```

```
NC_001422.1 5386 bp
Escherichia phage phiX174, complete genome
Escherichia phage phiX174 circular
32 features
```

Looping over features

Each feature has `.type` (`"gene"`, `"CDS"` ...), `.location` and `.qualifiers`, a dictionary whose values are **lists** of strings (a qualifier can appear more than once), so take item `[0]`. Use `.get()` with a default for qualifiers that may be missing:

```
from Bio import SeqIO

record = SeqIO.read("NC_001422.gbk", "genbank")
for feature in record.features:
    if feature.type != "CDS":
        continue
    tag = feature.qualifiers["locus_tag"][0]
    product = feature.qualifiers.get("product", ["unknown"])[0]
    print(tag, feature.location, len(feature), product)
```

```
phiX174p01 join{[3980:5386](+), [0:136](+)} 1542 DNA replication initiation
phiX174p02 join{[4496:5386](+), [0:136](+)} 1026 DNA replication initiation
phiX174p03 join{[5074:5386](+), [0:51](+)} 363 head morphogenesis
phiX174p04 [50:221](+) 171 K
phiX174p05 [132:393](+) 261 terminase
phiX174p07 [389:848](+) 459 head morphogenesis
phiX174p08 [567:843](+) 276 endolysin
phiX174p09 [847:964](+) 117 DNA condensation
phiX174p06 [1000:2284](+) 1284 major head protein
phiX174p10 [2394:2922](+) 528 major spike protein
phiX174p11 [2930:3917](+) 987 pilot protein for DNA ejection
```

Things to notice:

- Biopython locations are **0-based, end-exclusive** like Python slices: GenBank's `3981..5386` is printed `[3980:5386]`. `feature.location.start`, `.end` and `.strand` (`1` or `-1`) give the parts.
- The genome is circular, so the first three genes cross the origin: their location is a `join` of two pieces. `len(feature)` adds up the pieces.
- phiX174 packs 11 genes into 5.4 kb: several genes overlap or lie inside other genes (p02 is inside p01, in the same frame).

Extracting and translating a CDS

`feature.extract(record.seq)` returns the feature's sequence - joining the pieces and reverse complementing minus-strand features for you. Let's check it against the `/translation` that NCBI stored in the file:

```
from Bio import SeqIO

record = SeqIO.read("NC_001422.gbk", "genbank")
cds_list = [f for f in record.features if f.type == "CDS"]

first = cds_list[0]
print(first.location)
for key, values in first.qualifiers.items():
    print(" ", key, "=", values[0][:40])
```

```
cds_seq = first.extract(record.seq)      # follows the join() across the origin
print(len(cds_seq), cds_seq[:12], "...", cds_seq[-6:])
table = int(first.qualifiers["transl_table"][0])
protein = cds_seq.translate(table=table, cds=True)
print(protein[:30])
print(protein == first.qualifiers["translation"][0])
```

```
join{[3980:5386](+), [0:136](+)}
  locus_tag = phiX174p01
  function = viral strand synthesis
  note = rf replication
  codon_start = 1
  transl_table = 11
  product = DNA replication initiation
  protein_id = NP_040703.1
  db_xref = GeneID:2546398
  translation = MVRSYYPSECHADYFDFERIEALKPAIEACGISTLSQSPM
1542 ATGGTTCGTTCT ... AAATGA
MVRSYYPSECHADYFDFERIEALKPAIEAC
True
```

Writing all the proteins of a genome to a FASTA file is then a short loop:

```
from Bio import SeqIO
from Bio.SeqRecord import SeqRecord

record = SeqIO.read("NC_001422.gbk", "genbank")
proteins = []
for feature in record.features:
    if feature.type == "CDS":
        protein_seq = feature.extract(record.seq).translate(table=11, cds=True)
        proteins.append(SeqRecord(protein_seq,
                                  id=feature.qualifiers["protein_id"][0],
                                  description=feature.qualifiers["product"][0]))
print(SeqIO.write(proteins, "phiX174.pep.fasta", "fasta"), "proteins written")
```

```
11 proteins written
```

Converting between formats

`SeqIO.convert()` reads one format and writes another in a single step:

```
from Bio import SeqIO

count = SeqIO.convert("NC_001422.gbk", "genbank", "NC_001422.fasta", "fasta")
print("converted", count, "record")
```

```
converted 1 record
```

```
head -2 NC_001422.fasta
```

```
>NC_001422.1 Escherichia phage phiX174, complete genome
GAGTTTTATCGCTTCCATGACGCAGAAGTTAACACTTTCGGATATTTCTGATGAGTCGAA
```

You can only convert to a format that can hold the information: GenBank -> FASTA and FASTQ -> FASTA work (features or qualities are dropped), but FASTA -> FASTQ fails because there are no quality scores to write.

FASTQ files and quality scores

A FASTQ record has four lines: `@name`, the sequence, `+`, and one quality character per base. The character encodes a Phred score Q, the probability that the base call is wrong: $Q = 10$ means 1 in 10, $Q = 20$ means 1 in 100, $Q = 40$ means 1 in 10,000. Here is a tiny made-up file of four reads taken from the phiX174 sequence (in bash):

```
cat > reads.fastq << 'EOF'
@read1
CGAATTAAATCGAAGTGGACTGCTGGCGGA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII
@read2
AAACATTTGGACTGCTCCGCTTCCTCCTGA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII######
@read3
GCGGTCAAAAAGCCGCCTCCGGTGGCATTC
+
5555555555555555555555555555##
@read4
TTGACGGCCATAAGGCTGCTTCTGACGTTC
+
##############################
EOF
```

Biopython decodes the characters into numbers in `record.letter_annotations["phred_quality"]`:

```
from Bio import SeqIO

for record in SeqIO.parse("reads.fastq", "fastq"):
    quals = record.letter_annotations["phred_quality"]
    mean_q = sum(quals) / len(quals)
    print(record.id, quals[:3], quals[-3:], round(mean_q, 1))
```

```
read1 [40, 40, 40] [40, 40, 40] 40.0
read2 [40, 40, 40] [2, 2, 2] 32.4
read3 [20, 20, 20] [20, 2, 2] 18.8
read4 [2, 2, 2] [2, 2, 2] 2.0
```

(`I` is Q 40, `5` is Q 20 and `#` is Q 2 in the standard "Sanger"/Illumina 1.8+ encoding, which is Biopython's `"fastq"`.) Filtering reads, trimming them, and converting to FASTA:

```
from Bio import SeqIO

good = []
for record in SeqIO.parse("reads.fastq", "fastq"):
    quals = record.letter_annotations["phred_quality"]
    if sum(quals) / len(quals) >= 20:
        good.append(record)
print(SeqIO.write(good, "reads.good.fastq", "fastq"), "reads kept")

# trim read2 where the quality drops to 2 (the '#' characters)
read2 = good[1]
cut = read2.letter_annotations["phred_quality"].index(2)
print(read2[:cut].format("fastq"))

# FASTQ -> FASTA (the quality scores are dropped)
print(SeqIO.convert("reads.fastq", "fastq", "reads.fasta", "fasta"), "converted")
```

```
2 reads kept
@read2
AAACATTTGGACTGCTCCGCTTCC
+
IIIIIIIIIIIIIIIIIIIIIIII

4 converted
```

Slicing a `SeqRecord` (`read2[:cut]`) slices the qualities along with the sequence. This is fine for learning and for a few thousand reads; for real sequencing runs with millions of reads, use dedicated tools (`fastp`, `seqkit`), which are much faster.

Entrez: download from NCBI

`Bio.Entrez` talks to NCBI's E-utilities: **esearch** finds the IDs of records that match a query (the same query language as the NCBI web search box), and **efetch** downloads records by ID or accession. NCBI's rules:

- **Set `Entrez.email`** to your real email address, so NCBI can contact you if your script causes problems instead of blocking your IP (the whole cluster shares a few IP addresses!).
- **At most 3 requests per second** (10 with a free NCBI API key, set as `Entrez.api_key`). Biopython waits between requests for you, but don't write loops that fetch thousands of records one at a time - fetch many IDs in one request (`id=["NC_001422", "NC_001416"]`) and run large jobs outside US daytime.
- **Save what you download** to a file and re-use it (as in the `os.path.exists()` check above) rather than downloading again every time the script runs.

A search, then a fetch of the results:

```
import time
from Bio import Entrez, SeqIO

Entrez.email = "your.name@ucr.edu"

# 1. search: which protein records match?
handle = Entrez.esearch(db="protein", retmax=5,
                        term="ADH4[Gene Name] AND Saccharomyces cerevisiae[Organism]")
result = Entrez.read(handle)
handle.close()
print(result["Count"], "matches; first IDs:", result["IdList"])

time.sleep(1)       # be polite: no more than 3 requests per second

# 2. fetch: download those records in FASTA format and parse them
handle = Entrez.efetch(db="protein", id=result["IdList"][:3],
                       rettype="fasta", retmode="text")
for record in SeqIO.parse(handle, "fasta"):
    print(record.id, len(record), record.description[:50])
handle.close()
```

```
115 matches; first IDs: ['285811963', '269970305', '206558328', '205831682', '2863446362']
DAA07863.1 382 DAA07863.1 TPA: alcohol dehydrogenase ADH4 [Saccha
NP_011258.2 382 NP_011258.2 alcohol dehydrogenase ADH4 [Saccharomy
sp|A6ZTT5.2|ADH4_YEAS7 382 sp|A6ZTT5.2|ADH4_YEAS7 RecName: Full=Alcohol dehyd
```

Your count and IDs will differ - NCBI grows every day. Adding `AND refseq[filter]` to the query keeps only the curated RefSeq records. Common `db` values are `nuccore` (DNA/RNA), `protein`, `gene`, `taxonomy` and `pubmed`; common `rettype`s are `fasta`, `gb` (GenBank) and `gbwithparts` (GenBank including the sequence for big records).

To download whole genomes or proteomes, NCBI's `datasets` command-line tool or the NCBI FTP site (as in the BLAST lecture) is better than Entrez.

Pairwise alignment

`Bio.Align.PairwiseAligner` does global (Needleman-Wunsch) and local (Smith-Waterman) alignment of two sequences:

```
from Bio import Align

aligner = Align.PairwiseAligner()        # global alignment, match=1, mismatch=0
aligner.mismatch_score = -1
aligner.gap_score = -2
alignments = aligner.align("GATTACAGATTACA", "GATCACAGTTACA")
best = alignments[0]
print("score:", best.score)
print(best)
```

```
score: 9.0
target            0 GATTACAGATTACA 14
                  0 |||.||||-||||| 14
query             0 GATCACAG-TTACA 13
```

The Sequence evolution lecture uses it with BLOSUM62 to test whether two proteins are more similar than chance.

Other parts of Biopython

- `Bio.SeqUtils` - GC content, molecular weight, six-frame translations
- `Bio.AlignIO` / `Bio.Align` - read and write multiple sequence alignments (Clustal, FASTA, Stockholm, PHYLIP)
- `Bio.Phylo` - read, draw and edit phylogenetic trees (Newick, Nexus)
- `Bio.Blast` - run BLAST at NCBI, or read BLAST XML output (for `-outfmt 6` tables, use `csv` or pandas)
- `Bio.PDB` - protein 3D structures
- GFF3 files: Biopython itself doesn't read GFF (the separate `bcbio-gff` package does). A GFF3 file is a 9-column tab-separated table, so it is usually easiest to read it with pandas or with `csv` as in the SNP workshop.

Other packages for genomics

You will meet these in the workshops and in your projects. All install with `pip` or from Bioconda.

Package	For	Used in
pandas	tables: filter, group, join	Pandas, most workshops
matplotlib, seaborn	plots	Plotting
numpy, scipy	arrays, statistics	SNP workshop, Sequence evolution
pysam	read SAM/BAM/CRAM alignments, VCF/BCF, indexed FASTA	read alignment and variant data
cyvcf2	fast VCF/BCF reading	SNP workshop
pybedtools	bedtools from Python: overlaps of genomic intervals	Ranges and features
networkx	graphs and networks	Network workshop
duckdb	SQL queries on CSV/Parquet files	SQL with DuckDB

Common mistakes

- `ModuleNotFoundError`: the package isn't installed in *this* Python. Did you `conda activate` your environment (also inside your SLURM script)? Is the import name right (`import Bio`, not `import biopython`)?
- **Naming your own script after a module**, e.g. `random.py`, `csv.py` or `Bio.py`. `import random` then loads *your* file instead of the real module and you get strange errors such as `AttributeError: module 'random' has no attribute 'seed'`. Rename your file (and delete any `__pycache__` folder next to it).
- **Opening a `.gz` file with `open()`** or handing it straight to `SeqIO.parse()`: `UnicodeDecodeError: 'utf-8' codec can't decode byte 0x8b`. Use `gzip.open(f, "rt")`.
- **`gzip.open(f)` without `"rt"`** gives bytes: lines look like `b'>YAL001C\n'` and `line.startswith(">")` raises a `TypeError`.
- **`SeqIO.read()` on a file with many records**: `ValueError: More than one record found in handle`. Use `SeqIO.parse()` and a loop.
- **`SeqIO.index()` on a `.gz` file**: `Gzipped files are not suitable for indexing`. Uncompress it first (or use `bgzip`).
- **Off-by-one coordinates**: GFF/GenBank/VCF are 1-based and inclusive, BED files and Python are 0-based and end-exclusive. A 1-based `start..end` is `seq[start-1:end]`.
- **Forgetting the genetic code**: translating a mitochondrial or bacterial gene with the default table. Look at `/transl_table` in the GenBank file.
- **`subprocess.run()` without `check=True`**: the tool failed but your script continued and wrote empty results.
- **Unseeded random numbers**: results you can never reproduce exactly.
- **Downloading inside a loop**: hundreds of requests to NCBI get your IP blocked. Fetch in batches and save the files.
- **Old code from the web**: `Bio.Alphabet`, `GC()`, `open(f, "rU")` and `source activate` no longer work. Use no alphabet, `gc_fraction()`, `open(f)` and `conda activate`.

Quick reference

Task	Code
import a module	`import gzip`; `from Bio import SeqIO`; `import pandas as pd`
list files	`Path(".").glob("*.fasta")`
join a path	`Path("results") / "out.tsv"` or `os.path.join("results", "out.tsv")`
does the file exist?	`Path(f).exists()` or `os.path.exists(f)`
make a folder	`Path("results").mkdir(parents=True, exist_ok=True)`
read/write a .gz file	`gzip.open(f, "rt")` / `gzip.open(f, "wt")`
read a TSV without a header	`csv.DictReader(fh, fieldnames=cols, delimiter="\t")`
mean, median, sd	`statistics.mean(x)`, `.median(x)`, `.stdev(x)`
reproducible random numbers	`random.seed(42)`
count things	`Counter(items)`, `.most_common(5)`
group things	`d = defaultdict(list)`; `d[key].append(value)`
download a file	`urllib.request.urlretrieve(url, filename)`
run a program	`subprocess.run(["blastp", "-query", q], check=True)`
command-line options	`argparse.ArgumentParser()`, `add_argument()`, `parse_args()`
make an environment	`conda create -n NAME pkgs...` / `python3 -m venv DIR`

Task	Code
save the package list	`conda env export --from-history > environment.yml` / `pip freeze > requirements.txt`
sequence object	`Seq("ATG...")`, `.reverse_complement()`, `.translate(table=11)`
read sequences	`for rec in SeqIO.parse(fh, "fasta"):` (`"fastq"`, `"genbank"`)
read one record	`SeqIO.read(f, "genbank")`
write sequences	`SeqIO.write(records, "out.fasta", "fasta")`
convert formats	`SeqIO.convert(infile, "genbank", outfile, "fasta")`
look up by ID	`SeqIO.to_dict(SeqIO.parse(...))` or `SeqIO.index(f, "fasta")`
features	`rec.features`, `f.type`, `f.location`, `f.qualifiers["product"][0]`, `f.extract(rec.seq)`
quality scores	`rec.letter_annotations["phred_quality"]`
fetch from NCBI	`Entrez.email = ...`; `Entrez.efetch(db=, id=, rettype=, retmode="text")`

Exercises

1. Write a script that uses `pathlib` to list every `.gz` file in your data folder with its size in megabytes (`f.stat().st_size / 1e6`), sorted from largest to smallest.
2. Using `gzip` and `collections.Counter`, count the four bases (and anything else, such as `N`) in the whole yeast genome `S_cerevisiae.fasta.gz`. What is the GC content? Then do it per chromosome - which has the lowest GC content (hint: look at `chrmt`)?
3. Using `random` with a seed, simulate a 1 Mb random DNA sequence with the same GC content as yeast. How many `ATG`s and how many stop codons (`TAA`, `TAG`, `TGA`) do you expect per strand? Count them and compare. Run it again with the same seed and a different seed.
4. Use Biopython to read `S_cerevisiae.ORFs.fasta.gz`, translate every ORF, and report how many proteins are longer than 1000 amino acids. How many ORFs don't end in a stop codon, or have an internal stop (hint: `translate()` and look for `*` before the end; remember `chrmt` genes use table 3 - the ORF description tells you the chromosome)?
5. Turn exercise 4 into a proper command-line tool with `argparse`: an input FASTA file, `--min-aa` (default 100), `--table` (default 1) and `-o` for the output protein FASTA. Check that `-h` works and that bad values give a clear error.
6. Use `Entrez.efetch` to download the GenBank record for bacteriophage lambda (`NC_001416`) and save it. How many CDS features are there? Which is the longest? How many are on each strand? Write all the proteins to a FASTA file named by `protein_id`, and check your translations against the `/translation` qualifiers.
7. Using `subprocess`, run `blastp` of the phiX174 proteins you wrote above (`phiX174.pep.fasta`) against the lambda proteins from exercise 6, reading the results back into Python. Are any proteins similar between these two phages? (Make the database with `makeblastdb` from Python too.)
8. Challenge: write a script that takes a GenBank file and writes (a) a FASTA file of all CDS sequences (DNA), (b) a FASTA file of the proteins and (c) a tab-separated table with locus tag, start (1-based), end, strand, length and product for each CDS. Test it on the phiX174 and lambda records. Then read the table with pandas and find the mean CDS length.

Next

- Pandas: the second half of Python IV - reading, filtering, grouping and joining tables such as BLAST

results and GFF annotations.
- Python V: Regular expressions for finding patterns in text and sequences.
- Plotting with matplotlib and seaborn.