Python II: Decisions, lists, loops and files

In Python I we worked with one value at a time: one sequence, one line of a GFF file. Real data has thousands or millions of lines. This lecture gives you the tools to process a whole file: test a condition and act on it, keep many values in a list, repeat an action for every item, and read and write files.

What you'll learn

- `True`/`False`, comparisons (`==`, `<`, ...), `and`/`or`/`not`, and `in`
- making decisions with `if`/`elif`/`else`, and why indentation matters
- lists: making them, indexing and slicing, adding and removing items, sorting
- `for` loops over lists, strings and `range()`; `enumerate()` and `zip()`; `while` loops; `break` and `continue`
- the *accumulator* patterns for counting, summing and finding a maximum, and simple list comprehensions
- reading a file line by line, splitting it into columns and skipping headers
- writing results to a new file
- reading gzip-compressed files, and CSV files with the `csv` module
- getting file names from the command line with `sys.argv`

We will use these on real files: exon lengths from a BED file, genes from the yeast GFF annotation, the GC content of all yeast ORFs, and the IUCN list of threatened species.

True, False and comparisons

A **boolean** (`bool`) is either `True` or `False`. Comparisons produce booleans:

Operator	Meaning	Example	Result
`==`	equal to	`3 == 3`	`True`
`!=`	not equal to	`3 != 3`	`False`
`<`	less than	`2 < 3`	`True`
`<=`	less than or equal to	`3 <= 3`	`True`
`>`	greater than	`2 > 3`	`False`
`>=`	greater than or equal to	`2 >= 3`	`False`
`in`	is contained in	`"GC" in "AGCT"`	`True`

```
>>> length = 1812
>>> length > 1000
True
>>> length % 3 == 0
True
>>> strand = "+"
>>> strand == "-"
False
>>> strand != "-"
True
```

`=` *assigns* a value; `==` *compares* two values. Mixing them up is the most common mistake with `if` statements.

Strings are compared letter by letter, so `==` is exact: case and spaces count:

```
>>> "chrI" == "chrI"
True
>>> "chrI" == "chri"
False
>>> "chrI" == "chrI\n"
False
>>> "10" == 10
False
```

```
>>> "10" < "9"
True
>>> 10 < 9
False
```

The last three are the classic file-reading bugs: a line read from a file still has its newline, and a number read from a file is still a string. Comparing strings with `<` also goes letter by letter ("alphabetical"): `"1"` comes before `"9"`, so `"10" < "9"` is `True`. Convert to `int` or `float` before comparing numbers.

`in` tests whether a substring is inside a string (and, below, whether an item is in a list):

```
>>> seq = "ATGGCGTAGCTTAG"
>>> "TAG" in seq
True
>>> "N" in seq
False
>>> "N" not in seq
True
```

Combining conditions: `and`, `or`, `not`

- `a and b` is `True` only if both `a` and `b` are true
- `a or b` is `True` if at least one of them is true
- `not a` flips `True` to `False` and back

```
>>> length = 1812
>>> strand = "+"
>>> length > 1000 and strand == "+"
True
>>> length > 5000 and strand == "+"
False
>>> length > 5000 or strand == "+"
True
>>> not strand == "+"
False
>>> 1000 <= length <= 2000
True
```

The last one is a Python shortcut for `1000 <= length and length <= 2000`. Use parentheses to make complicated conditions clear, for example `(chrom == "chrI" or chrom == "chrII") and strand == "+"`.

Truthiness. In an `if`, values that aren't booleans count as false if they are "empty": `0`, `0.0`, `""` (the empty string), `[]` (the empty list) and `None`. Everything else counts as true. So `if line:` means "if the line is not empty".

To check whether a variable is `None`, use `is None` (or `is not None`). Don't use `is` for anything else - use `==` to compare numbers and strings.

Making decisions: `if`, `elif`, `else`

An `if` statement runs a block of code only when its condition is true:

```
length = 1237
if length % 3 != 0:
    print("warning: length is not a multiple of 3")
    print(f"{length % 3} extra bases")
print("done")
```

```
warning: length is not a multiple of 3
1 extra bases
done
```

The line with `if` ends with a **colon** `:`. The lines *indented* under it are the **block** that runs when the condition is true. The block ends at the first line that is not indented (`print("done")` always runs). This is different from bash, which uses `then` and `fi`: in Python, **indentation is the syntax**. Use 4 spaces per level (your editor will do this for you when you press Tab in a `.py` file).

Add `else:` for what to do otherwise, and `elif:` ("else if") to test more conditions in order. Only the *first* true branch runs:

```
gc = 0.62
if gc > 0.6:
    print("GC-rich")
elif gc < 0.4:
    print("AT-rich")
else:
    print("average GC")
```

```
GC-rich
```

Blocks can be nested inside each other. Each level is indented 4 more spaces:

```
seq = "ATGGCGTAGCTTAG"
if seq.startswith("ATG"):
    print("starts with a start codon")
    if seq[-3:] in ["TAA", "TAG", "TGA"]:
        print("and ends with a stop codon")
else:
    print("no start codon")
```

```
starts with a start codon
and ends with a stop codon
```

(`["TAA", "TAG", "TGA"]` is a list; lists are next.)

Common mistakes with `if`

- Forgetting the colon: `if gc > 0.6` is a `SyntaxError: expected ':'`.
- Inconsistent indentation. Each line of a block must be indented by exactly the same amount. A line indented for no reason gives `IndentationError: unexpected indent`, and forgetting to indent gives `IndentationError: expected an indented block after 'if' statement on line 2`.
- Mixing tabs and spaces. They look the same on screen but not to Python. Set your editor to insert spaces (Jupyter, VS Code and `nano` on the cluster can all do this), and if you get strange indentation errors, delete the indentation and retype it.
- `if strand = "+":` - one `=` is assignment; comparison is `==`.

Lists

A **list** holds many values in order. Write it with square brackets and commas:

```
>>> genes = ["YFG1", "CDC11", "SOD1"]
>>> lengths = [1812, 1248, 465]
>>> empty = []
>>> len(genes)
3
```

A list can hold any type, including a mix of types, but usually all of the items are the same kind of thing.

Indexing and slicing

Lists are indexed and sliced exactly like strings (Python I): the first item is `[0]`, the last is `[-1]`, and `[start:end]` doesn't include `end`:

```
>>> genes[0]
'YFG1'
>>> genes[-1]
'SOD1'
>>> genes[0:2]
['YFG1', 'CDC11']
>>> "CDC11" in genes
True
>>> "ACT1" in genes
False
```

Changing a list

Unlike strings, lists are **mutable**: you can change them in place.

Method	What it does
`lst[i] = x`	replace the item at index `i`
`lst.append(x)`	add `x` to the end
`lst.extend(other)`	add every item of the list `other` to the end
`lst.insert(i, x)`	insert `x` before index `i`
`lst.remove(x)`	remove the first item equal to `x`
`lst.pop()`	remove and return the last item (`pop(i)`: item `i`)
`lst.index(x)`	index of the first item equal to `x`
`lst.count(x)`	how many items are equal to `x`

```
>>> genes = ["YFG1", "CDC11", "SOD1"]
>>> genes.append("ACT1")
>>> genes
['YFG1', 'CDC11', 'SOD1', 'ACT1']
>>> genes.extend(["TUB1", "TUB2"])
>>> genes
['YFG1', 'CDC11', 'SOD1', 'ACT1', 'TUB1', 'TUB2']
>>> genes.insert(0, "HO")
>>> genes
['HO', 'YFG1', 'CDC11', 'SOD1', 'ACT1', 'TUB1', 'TUB2']
>>> genes.remove("SOD1")
>>> last = genes.pop()
>>> last
'TUB2'
>>> genes
['HO', 'YFG1', 'CDC11', 'ACT1', 'TUB1']
>>> genes[1] = "YFG2"
>>> genes
['HO', 'YFG2', 'CDC11', 'ACT1', 'TUB1']
```

`append` adds *one* item; `extend` adds each item of another list. If you `append` a list you get a list inside a list:

```
>>> a = [1, 2]
>>> a.append([3, 4])
>>> a
[1, 2, [3, 4]]
```

Numbers in lists: `sum`, `min`, `max`

```
>>> lengths = [1812, 1248, 465, 2210, 903]
>>> sum(lengths)
6638
>>> min(lengths)
465
>>> max(lengths)
2210
>>> sum(lengths) / len(lengths)
1327.6
```

These only work on numbers. A list of strings like `["1812", "1248"]` has to be converted first (see list comprehensions).

Sorting: `sorted()` and `.sort()`

There are two ways to sort. `sorted(lst)` *returns a new sorted list* and leaves the original alone. `lst.sort()` *sorts the list in place* and returns `None`. Both take `reverse=True` to sort from largest to smallest:

```
>>> lengths = [1812, 1248, 465, 2210, 903]
>>> sorted(lengths)
[465, 903, 1248, 1812, 2210]
>>> lengths
[1812, 1248, 465, 2210, 903]
>>> sorted(lengths, reverse=True)
[2210, 1812, 1248, 903, 465]
>>> lengths.sort()
>>> lengths
[465, 903, 1248, 1812, 2210]
```

A very common mistake is `lengths = lengths.sort()`, which sets `lengths` to `None`. Strings sort alphabetically, with all uppercase letters before lowercase:

```
>>> sorted(["chrX", "chrI", "chrV", "chrII"])
['chrI', 'chrII', 'chrV', 'chrX']
>>> sorted(["b", "a", "C", "B"])
['B', 'C', 'a', 'b']
>>> sorted(["chr2", "chr10", "chr1"])
['chr1', 'chr10', 'chr2']
```

The last one is the same problem as `"10" < "9"`: text is sorted character by character, so `chr10` comes before `chr2`.

Lists of lists

An item of a list can itself be a list. A table can be stored as a list of rows, where each row is a list of columns. `table[1]` is the second row and `table[1][0]` is the first column of that row:

```
>>> table = [["YFG1", 1812], ["CDC11", 1248], ["SOD1", 465]]
>>> table[1]
['CDC11', 1248]
>>> table[1][0]
'CDC11'
```

Lists of lists sort by their first item, then by the second item if the first ones are equal. So to sort genes by length, put the length first:

```
>>> rows = [[1812, "YFG1"], [1248, "CDC11"], [465, "SOD1"]]
>>> sorted(rows, reverse=True)
```

```
[[1812, 'YFG1'], [1248, 'CDC11'], [465, 'SOD1']]
```

(You will also see `sorted(table, key=lambda row: row[1])`, which means "sort the rows by column 1". The `lambda` is a small function; functions are in Python III.)

Strings and lists

`split()` makes a list from a string, and `join()` makes a string from a list of strings. `list()` turns a string into a list of its characters:

```
>>> "Chr7\t21408673\t21408826".split("\t")
['Chr7', '21408673', '21408826']
>>> list("ATGC")
['A', 'T', 'G', 'C']
>>> ",".join(["YFG1", "CDC11", "SOD1"])
'YFG1,CDC11,SOD1'
```

`join()` only works on strings. To join numbers, convert them first: `",".join([str(x) for x in lengths])` (list comprehensions are below).

Loops

`for` loops

A `for` loop runs its block once for each item in a list (or each character in a string, or each line in a file). Each time around, the loop variable is set to the next item:

```
genes = ["YFG1", "CDC11", "SOD1"]
for gene in genes:
    print("gene:", gene)
print("finished")
```

```
gene: YFG1
gene: CDC11
gene: SOD1
finished
```

As with `if`, the line ends with `:` and the block is indented. Compare with bash:

```
for gene in YFG1 CDC11 SOD1; do
    echo "gene: $gene"
done
```

The loop variable name (`gene`) is your choice; pick a name that says what one item is.

Looping over a string gives you one character at a time. Here we count bases with an `if` inside the loop:

```
seq = "ATGGCGTAGCTTAGNN"
gc = 0
other = 0
for base in seq:
    if base == "G" or base == "C":
        gc += 1
    elif base not in "AT":
        other += 1
print(f"G+C: {gc}  not ACGT: {other}  length: {len(seq)}")
```

```
G+C: 7  not ACGT: 2  length: 16
```

(`seq.count("G") + seq.count("C")` is quicker for this, but the loop shows the pattern that works for any question you want to ask about each base.)

Counting with `range()`

`range()` produces a sequence of whole numbers, like `seq` in bash. `range(n)` counts from 0 up to but *not including* `n`; `range(start, stop, step)` gives you more control. Wrap it in `list()` to see the numbers:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(1, 6))
[1, 2, 3, 4, 5]
>>> list(range(0, 20, 5))
[0, 5, 10, 15]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

`range()` with a step of 3 is exactly what you need to walk along a coding sequence one codon at a time:

```
seq = "ATGGCGTACGCTTAG"
for i in range(0, len(seq), 3):
    codon = seq[i:i+3]
    print(i, codon)
```

```
0 ATG
3 GCG
6 TAC
9 GCT
12 TAG
```

`enumerate()`: the index and the item

If you need to know *where* you are in the list as well as the item, use `enumerate()`. It gives you pairs of (index, item), which you unpack into two loop variables:

```
genes = ["YFG1", "CDC11", "SOD1"]
for i, gene in enumerate(genes):
    print(i, gene)
```

```
0 YFG1
1 CDC11
2 SOD1
```

Add `start=1` to count from 1: `enumerate(genes, start=1)`. This is cleaner than `for i in range(len(genes)):` followed by `genes[i]`, which you will see in older code.

`zip()`: two lists side by side

`zip()` walks through two (or more) lists at the same time:

```
genes = ["YFG1", "CDC11", "SOD1"]
lengths = [1812, 1248, 465]
for gene, length in zip(genes, lengths):
    print(f"{gene}\t{length}")
```

```
YFG1    1812
CDC11   1248
SOD1    465
```

`zip()` is also a neat way to compare two aligned sequences position by position:

```
seq1 = "ATGGCGTACGCT"
seq2 = "ATGGCTTACGAT"
diffs = 0
```

```
for a, b in zip(seq1, seq2):
    if a != b:
        diffs += 1
print(f"{diffs} differences in {len(seq1)} bp")
```

```
2 differences in 12 bp
```

`while` loops

A `while` loop repeats as long as its condition is true. Use it when you don't know in advance how many times to loop:

```
seq = "ATGAAATTTGGGTAGCCC"
i = 0
while i < len(seq) and seq[i:i+3] != "TAG":
    i += 3
print(f"stop codon at position {i}")
```

```
stop codon at position 12
```

Make sure something in the loop changes the condition, or it will run forever (press Ctrl-C to stop it). Without the `i < len(seq)` test, a sequence with no `TAG` would loop forever, because a slice past the end is just the empty string `""`, which is never equal to `"TAG"`. For looping over a list, string or file, a `for` loop is almost always simpler and safer.

`break` and `continue`

`continue` skips the rest of the block and goes on to the next item. `break` leaves the loop completely. Here we read codons until we reach a stop codon, skipping any codon that contains an `N`:

```
seq = "ATGNNNGCGTACTAGGCC"
stops = ["TAA", "TAG", "TGA"]
for i in range(0, len(seq), 3):
    codon = seq[i:i+3]
    if "N" in codon:
        print(i, codon, "skipped")
        continue
    if codon in stops:
        print(i, codon, "stop - done")
        break
    print(i, codon)
```

```
0 ATG
3 NNN skipped
6 GCG
9 TAC
12 TAG stop - done
```

The codon `GCC` after the stop was never looked at.

Accumulating results

Most data processing loops follow the same few patterns: set up a variable *before* the loop, update it *inside* the loop, and use it *after* the loop.

```
lengths = [1812, 1248, 465, 2210, 903, 150]

count = 0          # how many
total = 0          # a running sum
longest = 0        # the biggest so far
```

```
long_genes = []   # a new list of the items we want to keep
for length in lengths:
    count += 1
    total += length
    if length > longest:
        longest = length
    if length >= 1000:
        long_genes.append(length)

print(f"n={count} total={total} mean={total / count:.1f} max={longest}")
print(f"{len(long_genes)} genes >= 1000 bp: {long_genes}")
```

```
n=6 total=6788 mean=1131.3 max=2210
3 genes >= 1000 bp: [1812, 1248, 2210]
```

For a plain list, `len()`, `sum()` and `max()` would do this for you, but when you are reading a file line by line you usually don't have a list - you accumulate as you go. Starting `longest` at 0 works for lengths, which can't be negative. For data that could be negative, start with the first value instead, or use `None` and check for it.

List comprehensions

A **list comprehension** builds a new list from an old one in one line. These two pieces of code do the same thing:

```
fields = ["1812", "1248", "465"]

numbers = []
for x in fields:
    numbers.append(int(x))
print(numbers)

numbers = [int(x) for x in fields]
print(numbers)
```

```
[1812, 1248, 465]
[1812, 1248, 465]
```

Read `[int(x) for x in fields]` as "a list of `int(x)` for each `x` in `fields`". You can add an `if` at the end to keep only some items:

```
lengths = [1812, 1248, 465, 2210, 903, 150]
print([x for x in lengths if x >= 1000])
print([x // 3 for x in lengths])
genes = ["yfg1", "cdc11", "sod1"]
print([g.upper() for g in genes])
```

```
[1812, 1248, 2210]
[604, 416, 155, 736, 301, 50]
['YFG1', 'CDC11', 'SOD1']
```

Use comprehensions for simple one-step transformations like these. If you need more than one `if` or several steps, write a normal `for` loop - it is easier to read.

Reading files

Get the data

Make a folder for today and download the data files into it:

```
mkdir -p ~/bigdata/gen220/python2
cd ~/bigdata/gen220/python2
```

```
GEN220=https://raw.githubusercontent.com/biodataprog/GEN220/master
curl -sSLO $GEN220/data/rice_random_exons.bed
URL=https://github.com/biodataprog/GEN220_data/raw/main
curl -sSLO $URL/genome/S_cerevisiae.gff3.gz
curl -sSLO $URL/genome/S_cerevisiae.ORFs.fasta.gz
curl -sSLO $URL/tabular/threatened-species.csv.gz
```

Run the examples below from this folder (in the terminal with `python3 script.py`, or in a Jupyter notebook started in this folder). `rice_random_exons.bed` is a BED file with three tab-separated columns: chromosome, start and end of 1000 rice exons.

```
head -n 3 rice_random_exons.bed
```

```
Chr7    21408673    21408826
Chr9    16031526    16031938
Chr11   4762531 4762595
```

Opening a file and reading it line by line

`open(filename)` opens a file and gives you a **file handle**, an object you read from. The best way to use it is in a `with` block, which closes the file automatically when the block ends (even if there is an error). A `for` loop over the file handle gives you one line at a time:

```
with open("rice_random_exons.bed") as fh:
    for line in fh:
        print(repr(line))
        break
```

```
'Chr7\t21408673\t21408826\n'
```

We used `repr()` to see the hidden characters, and `break` to stop after the first line. Each line is a **string** that still ends with the newline `\n`. Before using a line, nearly always:

1. `line.strip()` - remove the newline (and any stray spaces) from the ends
2. `.split("\t")` - split the columns at each tab to get a list
3. convert the columns you need to numbers with `int()` or `float()`

```
with open("rice_random_exons.bed") as fh:
    for line in fh:
        cols = line.strip().split("\t")
        chrom = cols[0]
        start = int(cols[1])
        end = int(cols[2])
        print(chrom, start, end, end - start)
        break
```

```
Chr7 21408673 21408826 153
```

Reading one line at a time uses very little memory, so this works the same on a file with a billion lines. (There is also `fh.read()`, which reads the whole file into one string, and `fh.readlines()`, which makes a list of all lines. Avoid them for big files.)

The file name is a path, relative to the folder you are running Python in (like any UNIX command). If Python can't find the file you get `FileNotFoundError: [Errno 2] No such file or directory: 'rice_random_exons.bed'`. Check where you are with `pwd` and what is there with `ls`.

Summarizing a column: sum, mean, min and max

Putting the accumulator patterns together with the file loop gives a summary of all the exon lengths. In BED format the length is `end - start` (BED starts count from 0, so no `+ 1`; see Python I):

```python
#!/usr/bin/env python3
# bed_summary.py - count, total and mean length of the features in a BED file

count = 0
total = 0
longest = 0
shortest = None
with open("rice_random_exons.bed") as fh:
    for line in fh:
        cols = line.strip().split("\t")
        length = int(cols[2]) - int(cols[1])
        count += 1
        total += length
        if length > longest:
            longest = length
        if shortest is None or length < shortest:
            shortest = length

print(f"exons:    {count}")
print(f"total bp: {total}")
print(f"mean bp:  {total / count:.1f}")
print(f"shortest: {shortest}")
print(f"longest:  {longest}")
```

```
python3 bed_summary.py
```

```
exons:    1000
total bp: 369855
mean bp:  369.9
shortest: 13
longest:  5818
```

`shortest` starts as `None` ("no value yet"), so the first exon always becomes the shortest so far. You can check the answer in bash with `awk`, as in the UNIX III lab:

```
awk '{n++; t += $3 - $2} END {print n, t, t/n}' rice_random_exons.bed
```

```
1000 369855 369.855
```

If you need the lengths again later (to sort them, or find the median), collect them in a list as you read, and use `len()`, `sum()`, `min()`, `max()` and `sorted()` afterwards:

```python
lengths = []
with open("rice_random_exons.bed") as fh:
    for line in fh:
        cols = line.strip().split("\t")
        lengths.append(int(cols[2]) - int(cols[1]))

lengths.sort()
print(len(lengths), min(lengths), max(lengths))
print("median:", lengths[len(lengths) // 2])
print("five longest:", lengths[-5:])
```

```
1000 13 5818
median: 176
five longest: [3536, 3665, 3802, 3987, 5818]
```

(For an even number of values the true median is the mean of the two middle values; this is close enough for a quick look.)

Skipping headers, comments and blank lines

Most files have lines that aren't data: a header row with column names, comment lines starting with `#`, or blank lines. Test for them at the top of the loop and `continue`:

```
for line in fh:
    if line.startswith("#"):
        continue            # skip comments
    line = line.strip()
    if not line:
        continue            # skip blank lines
    ...
```

To skip a header row that is the first line of the file, call `next(fh)` once before the loop; it reads and throws away one line. Or check the line itself, `if line.startswith("gene_id"): continue`.

Compressed files: `gzip`

Genomics files are usually gzip-compressed (`.gz`). You don't need to uncompress them: the `gzip` module (part of the Python standard library) opens them for you. `import gzip` at the top of the script loads the module, and then `gzip.open()` is used just like `open()`. The `"rt"` means **r**ead **t**ext; without the `t` you get raw bytes instead of strings.

The yeast GFF3 annotation starts with comment lines, then has 9 tab-separated columns: chromosome, source, feature type, start, end, score, strand, phase and attributes.

```
import gzip

with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        if line.startswith("#"):
            continue
        cols = line.strip().split("\t")
        print(cols[0], cols[2], cols[3], cols[4], cols[6])
        break
```

```
chrI chromosome 1 230218 .
```

Filtering: genes on one chromosome

Now we can ask a real question: how many genes are on chromosome I, how many on each strand, and how long are they? GFF coordinates start at 1 and include the end, so the length is `end - start + 1`.

```
#!/usr/bin/env python3
# chrI_genes.py - count the genes on chromosome I of yeast
import gzip

plus = 0
minus = 0
total_length = 0
with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        if line.startswith("#"):
            continue
        cols = line.strip().split("\t")
        if cols[0] != "chrI" or cols[2] != "gene":
            continue
        length = int(cols[4]) - int(cols[3]) + 1
        total_length += length
```

```
        if cols[6] == "+":
            plus += 1
        else:
            minus += 1

genes = plus + minus
print(f"chrI genes: {genes} (+ strand {plus}, - strand {minus})")
print(f"mean gene length: {total_length / genes:.0f} bp")
```

```
python3 chrI_genes.py
```

```
chrI genes: 117 (+ strand 60, - strand 57)
mean gene length: 1258 bp
```

The line `if cols[0] != "chrI" or cols[2] != "gene": continue` says "skip anything that isn't a gene on chrI". Filtering *out* what you don't want with `continue` keeps the main part of the loop from being indented very deeply.

GC content of a FASTA file

A FASTA file has header lines starting with `>` followed by lines of sequence. To get the GC content of *all* the sequence in a file (a whole genome, or all the genes), skip the header lines and add up the counts from every sequence line:

```
#!/usr/bin/env python3
# gc_fasta.py - overall GC content of all the sequences in a FASTA file
import gzip

gc = 0
total = 0
n_seqs = 0
with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    for line in fh:
        if line.startswith(">"):
            n_seqs += 1
            continue
        seq = line.strip().upper()
        gc += seq.count("G") + seq.count("C")
        total += len(seq)

print(f"{n_seqs} sequences, {total:,} bp, GC = {gc / total:.2%}")
```

```
python3 gc_fasta.py
```

```
6713 sequences, 9,078,756 bp, GC = 39.61%
```

This treats the file as one long sequence. Computing the GC content of *each* sequence separately means keeping track of which sequence you are in; that is the FASTA parser in Python III. For an uncompressed file (`.fasta`, `.fna`), use `open()` instead of `gzip.open()`; everything else is the same.

Writing files

Open a file for writing with mode `"w"`. **This replaces the file if it already exists.** (Mode `"a"` *appends* to the end instead, like `>>` in bash.) The file handle's `write()` method writes a string. Unlike `print()`, `write()` does not add a newline or spaces for you, and it only accepts strings - so an f-string ending in `\n` is the easiest way to build each line:

```
#!/usr/bin/env python3
# bed_lengths.py - write a table of exon lengths, keeping exons >= 500 bp
```

```
n_written = 0
with open("rice_random_exons.bed") as fh, open("long_exons.tsv", "w") as out:
    out.write("chrom\tstart\tend\tlength\n")
    for line in fh:
        chrom, start, end = line.strip().split("\t")
        length = int(end) - int(start)
        if length >= 500:
            out.write(f"{chrom}\t{start}\t{end}\t{length}\n")
            n_written += 1
print(f"wrote {n_written} exons to long_exons.tsv")
```

```
python3 bed_lengths.py
head -n 4 long_exons.tsv
```

```
wrote 205 exons to long_exons.tsv
chrom   start   end length
Chr3    16171331    16172869    1538
Chr1    3667439 3668072 633
Chr3    15041535    15042398    863
```

Two new things here:

- One `with` can open several files, separated by commas.
- `chrom, start, end = line.strip().split("\t")` **unpacks** the list of three columns into three variables in one step. It only works if the line has exactly three columns; otherwise you get `ValueError: too many values to unpack` (or "not enough").

You can also use `print(..., file=out)`, which adds the newline for you and converts numbers to text: `print(chrom, start, end, length, sep="\t", file=out)`.

Always write a header line, and prefer tabs between columns: the output can then be read by `sort`, `cut`, `awk`, R, pandas or Excel.

Reading a table with a header, and sorting rows

Now read `long_exons.tsv` back in. Its first line is a header, so we skip it with `next(fh)`, which reads one line and throws it away. To sort the rows by length we keep each row as a small list with the length *first* (lists of lists sort by their first item; see Lists of lists):

```
rows = []
with open("long_exons.tsv") as fh:
    header = next(fh)
    for line in fh:
        chrom, start, end, length = line.strip().split("\t")
        rows.append([int(length), chrom, start, end])

print("header:", header.strip().split("\t"))
print("rows:", len(rows))
rows.sort(reverse=True)
for length, chrom, start, end in rows[:3]:
    print(f"{chrom}:{start}-{end}\t{length}")
```

```
header: ['chrom', 'start', 'end', 'length']
rows: 205
Chr2:17975289-17981107  5818
Chr1:14587767-14591754  3987
Chr5:948466-952268  3802
```

The `for` loop unpacks each row of four items into four variables, the same way we unpacked the columns of a line. Converting `length` to `int` before sorting matters: as strings, `"999"` would sort *after* `"5818"`.

CSV files and the csv module

`.split(",")` is fine for simple comma-separated files, but a CSV field that itself contains a comma is put in double quotes, and `split` doesn't know about quotes (see CSV quoting in the DuckDB lecture). The IUCN threatened species file has a header line, and the `taxonomic_authority` column sometimes contains commas:

```
zcat threatened-species.csv.gz | grep -m 1 '"' | cut -d, -f 8-10
```

```
Polyspora hirtella,"(Ridl.) Orel, Peter G.Wilson
```

(On a Mac use `zcat < file.gz` or `gzcat`.) We asked `cut` for columns 8 to 10 (scientific name, authority, infraspecific rank), but it split the quoted authority `"(Ridl.) Orel, Peter G.Wilson, ..."` at its commas.

The `csv` module reads quoted fields correctly. `csv.reader(fh)` gives you each line already split into a list of columns. Here we compare it with `split(",")` on the first line that has a quote in it. `cut` counts columns from 1, so its columns 8 to 10 are `[7:10]` in Python:

```
import csv
import gzip

with gzip.open("threatened-species.csv.gz", "rt") as fh:
    for line in fh:
        if '"' in line:
            break

cols = line.strip().split(",")
print(len(cols), cols[7:10])
row = next(csv.reader([line]))
print(len(row), row[7:10])
```

```
16 ['Polyspora hirtella', '"(Ridl.) Orel', ' Peter G.Wilson']
14 ['Polyspora hirtella', '(Ridl.) Orel, Peter G.Wilson, Curry &amp;amp; Luu', '']
```

(`&` is how the original web data wrote `&`.) `split(",")` finds 16 columns instead of 14 and breaks the authority into pieces, so every later column (such as the category) is in the wrong place. `csv.reader` gets 14. (`csv.reader` normally reads a file handle; here we gave it a list containing one line, and `next()` takes the first row.)

In a real script, give `csv.reader` the file handle and loop over the rows. Use `next()` once to take the header row. Let's count the critically endangered (`CR`) species in each kingdom (column 1; the category is column 12):

```
import csv
import gzip

animals = 0
plants = 0
other = 0
with gzip.open("threatened-species.csv.gz", "rt") as fh:
    reader = csv.reader(fh)
    header = next(reader)
    print(header[1], header[12])
    for row in reader:
        if row[12] != "CR":
            continue
        if row[1] == "ANIMALIA":
            animals += 1
        elif row[1] == "PLANTAE":
            plants += 1
        else:
```

```
            other += 1
print(f"critically endangered: {animals} animals, {plants} plants, {other} other")
```

```
kingdom_name category
critically endangered: 3136 animals, 5401 plants, 39 other
```

(Counting *every* category at once needs a dictionary; see Python III.) For tab-separated files use `csv.reader(fh, delimiter="\t")`. To write CSV, `csv.writer` adds the quotes for you when a value contains the delimiter:

```
import csv

with open("species.csv", "w", newline="") as out:
    writer = csv.writer(out)
    writer.writerow(["species", "authority", "category"])
    writer.writerow(["Polyspora hirtella", "(Ridl.) Orel, Peter G.Wilson", "DD"])
with open("species.csv") as fh:
    print(fh.read())
```

```
species,authority,category
Polyspora hirtella,"(Ridl.) Orel, Peter G.Wilson",DD
```

(`newline=""` is recommended by the `csv` documentation when writing, so line endings are handled correctly on every system.)

Command-line arguments: `sys.argv`

So far the file names have been written into the scripts. To make a script you can run on *any* file, like a UNIX command, read the file name from the command line. The `sys` module has a list called `sys.argv`: `sys.argv[0]` is the name of the script and `sys.argv[1]`, `sys.argv[2]`, … are the arguments after it (like `$0`, `$1`, `$2` in bash). They are always strings. This tiny script just prints them:

```
import sys
print(sys.argv)
print(len(sys.argv), "items; the first argument is", sys.argv[1])
```

```
python3 show_args.py rice_random_exons.bed 500
```

```
['show_args.py', 'rice_random_exons.bed', '500']
3 items; the first argument is rice_random_exons.bed
```

Note that `500` arrived as the string `'500'`; use `int(sys.argv[2])` if you need a number. Here is a more useful script, which summarizes any number of BED files:

```
#!/usr/bin/env python3
# bed_stats.py - summarize feature lengths in one or more BED files
# usage: bed_stats.py FILE.bed [FILE2.bed ...]
import sys

if len(sys.argv) < 2:
    print(f"usage: {sys.argv[0]} FILE.bed [FILE2.bed ...]", file=sys.stderr)
    sys.exit(1)

print("file\tfeatures\ttotal_bp\tmean_bp")
for filename in sys.argv[1:]:
    count = 0
    total = 0
    with open(filename) as fh:
        for line in fh:
            if line.startswith("#") or line.startswith("track"):
                continue
```

```
            cols = line.strip().split("\t")
            count += 1
            total += int(cols[2]) - int(cols[1])
    print(f"{filename}\t{count}\t{total}\t{total / count:.1f}")
```

```
chmod +x bed_stats.py
./bed_stats.py
head -n 100 rice_random_exons.bed > first100.bed
./bed_stats.py rice_random_exons.bed first100.bed
```

```
usage: ./bed_stats.py FILE.bed [FILE2.bed ...]
file    features    total_bp    mean_bp
rice_random_exons.bed   1000    369855  369.9
first100.bed    100 36702   367.0
```

- `sys.argv[1:]` is every argument after the script name, so the loop handles as many files as you give it (including a wildcard like `./bed_stats.py *.bed`).
- If there are no arguments, the script prints a **usage** message and stops with `sys.exit(1)`. A non-zero exit code tells bash that something went wrong (UNIX III).
- `file=sys.stderr` sends the message to STDERR, so it doesn't end up in your output file if you redirect STDOUT with `>`.

For scripts with many options (`-o output.txt`, `--min-length 500`), the `argparse` module is covered in Python IV.

Reading from a pipe. `sys.stdin` is a file handle that is already open for standard input, so a script can also read data piped into it, like any UNIX tool:

```
#!/usr/bin/env python3
# count_genes.py - count "gene" lines in GFF data read from STDIN
import sys

genes = 0
for line in sys.stdin:
    cols = line.split("\t")
    if len(cols) > 2 and cols[2] == "gene":
        genes += 1
print(genes, "genes")
```

```
zcat S_cerevisiae.gff3.gz | python3 count_genes.py
```

```
6600 genes
```

The `len(cols) > 2` test makes the script skip comment lines, which don't have enough columns.

Common mistakes

- **Forgetting to `strip()`.** The last column still has `\n` on the end, so `cols[2] == "gene"` is false when `cols[2]` is really `"gene\n"`.
- **Forgetting to convert.** `cols[1]` is the string `"21408673"`; `cols[2] - cols[1]` is a `TypeError`, and `"9" > "10"` is `True`.
- **Off-by-one.** BED: `length = end - start`; GFF/VCF: `length = end - start + 1`. `range(n)` stops at `n - 1`.
- **Wrong indentation.** A `print` indented inside the loop runs once per line; outdented, it runs once at the end. Put summary output *after* the loop.
- **Setting up the counter inside the loop.** `total = 0` inside the loop resets it on every line. Accumulators are created before the loop.
- `lst = lst.sort()` sets `lst` to `None`. Use `lst.sort()` or `lst = sorted(lst)`.

- **Opening the output with "w" when you meant to read.** `open("data.bed", "w")` empties the file immediately.
- **`write()` needs a string and a newline.** `out.write(length)` is a `TypeError`; use `out.write(f"{length}\n")`.
- **Splitting CSV with `split(",")`.** Use the `csv` module for files you didn't make.

Quick reference

Task	Code
compare	`==  !=  <  <=  >  >=`
combine conditions	`and`, `or`, `not`
substring / item in a list	`"ATG" in seq`, `"chrI" in chroms`
decisions	`if ...:` / `elif ...:` / `else:`
make a list	`x = []`, `x = [1, 2, 3]`, `list("ATG")`
add to a list	`x.append(item)`, `x.extend(other_list)`
remove from a list	`x.remove(item)`, `x.pop()`
sort	`sorted(x)`, `x.sort()`, `sorted(x, reverse=True)`
sum, min, max, length	`sum(x)`, `min(x)`, `max(x)`, `len(x)`
loop over items	`for item in x:`
loop over numbers	`for i in range(0, len(seq), 3):`
index and item	`for i, item in enumerate(x):`
two lists together	`for a, b in zip(x, y):`
loop while true	`while condition:`
skip to next / stop loop	`continue` / `break`
list comprehension	`[int(v) for v in cols if v != ""]`
read a file	`with open(name) as fh:` then `for line in fh:`
read a .gz file	`import gzip`; `gzip.open(name, "rt")`
clean and split a line	`cols = line.strip().split("\t")`
skip comments	`if line.startswith("#"): continue`
skip a header line	`next(fh)`
write a file	`with open(name, "w") as out:` then `out.write(f"...\n")`
CSV	`import csv`; `for row in csv.reader(fh):`
command-line arguments	`import sys`; `sys.argv[1]`, `sys.argv[1:]`
stop with an error	`sys.exit(1)`

Exercises

Use the files downloaded above. Write each answer as a script with comments.

1. **Codon check.** Given `seq = "ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG"`, use a `for` loop with `range()` to print every codon and its position. Then print whether the sequence starts with `ATG`, ends with a stop codon, and has a length that is a multiple of 3.
2. **Lists.** Start with `lengths = [1812, 1248, 465, 2210, 903, 150, 3001]`. Print the number of values, the mean, the three largest values (using `sorted`), and a new list with only the values between 500 and 2000.
3. **Differences.** For `s1 = "ATGCGTACGTTAGC"` and `s2 = "ATGCGAACGTTTGC"`, use `zip()` and `enumerate()` to print the position and the two bases at each difference, then the percent identity.
4. **BED filter.** From `rice_random_exons.bed`, how many exons are on `Chr1`, and what is their total length? Write the Chr1 exons longer than 300 bp to a new BED file `chr1_long.bed`. Check your answers with `awk` in bash.
5. **GFF feature types.** Count the number of `gene`, `tRNA_gene` and `snoRNA_gene` features in `S_cerevisiae.gff3.gz`, and the mean length of each. (Hint: three counters and three totals, and an `if`/`elif`.)

6. **Command-line GC.** Change `gc_fasta.py` so it takes one or more FASTA file names on the command line and prints a tab-delimited line for each file: file name, number of sequences, total bp, GC percent. Use `gzip.open()` if the name ends with `.gz`, otherwise `open()`.
7. **Threatened species.** Using the `csv` module, count how many species in the class `AMPHIBIA` are in each of the categories `CR`, `EN` and `VU`, and write the `scientific_name` and `category` of every `CR` amphibian to a new tab-delimited file with a header line.
8. **Binning.** Make a histogram of the rice exon lengths in 100 bp bins: how many exons are 0-99 bp, 100-199 bp, and so on. Write a two-column CSV file `bin,count` with one line per bin, where `bin` is the start of the bin (0, 100, 200, ...).

Hints

- Exercise 3: percent identity is (matches / length) * 100.
- Exercise 4: check the count with `awk '$1 == "Chr1"' rice_random_exons.bed | wc -l` and the total with `awk '$1 == "Chr1" {t += $3 - $2} END {print t}'`.
- Exercise 6: inside the loop over `sys.argv[1:]`, use `if filename.endswith(".gz"):` to choose between `fh = gzip.open(filename, "rt")` and `fh = open(filename)`. Then loop over `fh` as usual and call `fh.close()` when you are done with the file.
- Exercise 8: `length // 100` is the bin number (Python I) and `length // 100 * 100` is the start of the bin. First collect all the lengths in a list. Then make a list of counts with one zero per bin (`[0] * n_bins` makes a list of `n_bins` zeros, the same way `"N" * 10` makes a string of 10 Ns) - how many bins do you need, given the longest exon? Loop over the lengths and add 1 to the right count. Finally loop over the counts with `enumerate()` to write each `bin,count` line. In Python III you will see how a dictionary does the same job without needing to know the largest value first.

Next

Python III: dictionaries (look up a value by name, count things by category), sets, tuples, writing your own functions, and a FASTA parser that keeps each sequence separately.

Further reading

- Python tutorial: control flow, lists and reading and writing files
- The csv and gzip module documentation
- The list of built-in functions