Python IV: Tables with Pandas

Most biological data ends up as a table: BLAST hits, GFF annotations, BED intervals, read counts per gene, sample sheets, species lists. You can process these with loops, `split("\t")` and dictionaries (as in Python II and Python III), but every question needs a new loop. **pandas** gives you a table object, the **DataFrame**, with ready-made operations for the things you do over and over: filter rows, compute new columns, sort, count, group and summarize, and join two tables together.

What you'll learn

1. The two pandas objects: `Series` (one column) and `DataFrame` (a table)
2. Reading CSV/TSV files, compressed or from a URL, including files with no header (BLAST `-outfmt 6`, BED) and files with comment lines (GFF3)
3. Looking at a table: `head`, `shape`, `dtypes`, `info`, `describe`
4. Selecting columns and rows: `df["col"]`, boolean filters, `.loc` and `.iloc`
5. New columns, sorting, `value_counts`, `groupby` and aggregation
6. Joining two tables with `merge`
7. Missing values, and the "NA" trap
8. Reshaping (`crosstab`, `pivot_table`, `melt`), and writing CSV and Parquet
9. Worked examples: a yeast genome summary, the best BLAST hit per gene, threatened species counts
10. When to use pandas, a plain loop, SQL/DuckDB or `awk`

This is the second half of Python IV; the first half is Modules, packages and Biopython. The same kinds of questions are answered with SQL in Tabular data with DuckDB, and the Plotting lecture makes figures from pandas tables.

Setup

pandas is not part of the standard library. Install it in your environment (see Installing packages); `pyarrow` is needed for Parquet files:

```
conda install pandas pyarrow          # or: pip install pandas pyarrow
python -c "import pandas; print(pandas.__version__)"
```

Everyone imports it with the nickname `pd`:

```
import pandas as pd
```

The examples were run with pandas 3.0.6 (and pyarrow 25). With pandas 2.x everything works the same; the one difference you will see is that text columns are listed as `object` instead of `str` by `dtypes` and `info()`.

The examples in each section build on each other: run them one after the other in the same Python session (`ipython`, a Jupyter notebook), or add each piece to the end of one script. Scripts that start with `#!/usr/bin/env python3` stand on their own.

Data used in this lecture

```
mkdir -p ~/bigdata/python4
cd ~/bigdata/python4
BASE=https://github.com/biodataprog/GEN220_data/raw/main
curl -L -O $BASE/genome/S_cerevisiae.gff3.gz              # yeast annotation (GFF3)
curl -L -O $BASE/genome/Ecoli-vs-Senterica.BLASTP.tab.gz  # BLASTP, -outfmt 6
curl -L -O $BASE/genome/E_coli_K12.pep.gz                 # E. coli proteins
curl -L -O $BASE/tabular/threatened-species.csv.gz        # IUCN Red List table
curl -O https://raw.githubusercontent.com/biodataprog/GEN220/master/data/rice_random_exons.bed
```

Series and DataFrames

Series: one column with labels

A `Series` is a list of values with a label (the **index**) for each one. Here are the lengths of the eight RNA segments of influenza A virus (strain A/Puerto Rico/8/1934):

```
import pandas as pd

lengths = pd.Series([2341, 2341, 2233, 1778, 1565, 1413, 1027, 890],
                    index=["PB2", "PB1", "PA", "HA", "NP", "NA", "M", "NS"])
print(lengths)
print(lengths["HA"], lengths.max(), lengths.sum())
```

```
PB2    2341
PB1    2341
PA     2233
HA     1778
NP     1565
NA     1413
M      1027
NS      890
dtype: int64
1778 2341 13588
```

Arithmetic and comparisons work on **every value at once** - no loop needed. This is called *vectorized* code, and it is both shorter and much faster than a loop:

```
print(lengths / 1000)
print(lengths > 2000)
print(lengths[lengths > 2000])      # keep the values where the test is True
```

```
PB2    2.341
PB1    2.341
PA     2.233
HA     1.778
NP     1.565
NA     1.413
M      1.027
NS     0.890
dtype: float64
PB2     True
PB1     True
PA      True
HA     False
NP     False
NA     False
M      False
NS     False
dtype: bool
PB2    2341
PB1    2341
PA     2233
dtype: int64
```

DataFrame: a table

A `DataFrame` is a set of named columns (each one a Series) that share the same row index. You can make one from a dictionary of lists:

```
flu = pd.DataFrame({
    "segment": [1, 2, 3, 4, 5, 6, 7, 8],
    "gene":    ["PB2", "PB1", "PA", "HA", "NP", "NA", "M", "NS"],
    "length":  [2341, 2341, 2233, 1778, 1565, 1413, 1027, 890],
})
print(flu)
print(flu.shape)                # (rows, columns)
print(flu["length"].mean())     # one column is a Series
```

```
   segment gene  length
0        1  PB2    2341
1        2  PB1    2341
2        3   PA    2233
3        4   HA    1778
4        5   NP    1565
5        6   NA    1413
6        7    M    1027
7        8   NS     890
(8, 3)
1698.5
```

The numbers on the left (0-7) are the row index. When you read a file, pandas numbers the rows like this unless you tell it to use one of the columns as the index.

In practice you will almost always make DataFrames by reading files.

Reading files

CSV files, compressed or from the web

`pd.read_csv()` reads a comma-separated file with a header line. It uncompresses `.gz` (and `.bz2`, `.zip`, `.xz`) files automatically, and it can read directly from a URL:

```
url = ("https://github.com/biodataprog/GEN220_data/raw/main/"
       "tabular/threatened-species.csv.gz")
species = pd.read_csv(url)       # or the local file: "threatened-species.csv.gz"
print(species.shape)
print(species.columns.tolist())
```

```
(143732, 14)
['taxonid', 'kingdom_name', 'phylum_name', 'class_name', 'order_name', 'family_name', 'genus_name', 'scie
```

The IUCN Red List table has one row per assessed species, with its taxonomy and its Red List `category`: `CR` (critically endangered), `EN` (endangered), `VU` (vulnerable), `NT` (near threatened), `LC` (least concern), `DD` (data deficient), `EX` (extinct) and a few older categories (`LR/...`).

Tab-separated files with no header: BLAST output

BLAST `-outfmt 6` output (see the BLAST lecture) is tab-separated with 12 columns and **no header line**. Tell pandas three things: `sep="\t"` (tab separated), `header=None` (the first line is data, not column names) and `names=[...]` (the names to use). This file compares *E. coli* K-12 proteins (queries) with *Salmonella enterica* proteins (subjects):

```
blast_columns = ["qseqid", "sseqid", "pident", "length", "mismatch", "gapopen",
                 "qstart", "qend", "sstart", "send", "evalue", "bitscore"]
```

```
blast = pd.read_csv("Ecoli-vs-Senterica.BLASTP.tab.gz", sep="\t",
                    header=None, names=blast_columns)
print(blast.shape)
print(blast[["qseqid", "pident", "length", "evalue", "bitscore"]].head(3))
```

```
(20549, 12)
                         qseqid  pident  length         evalue  bitscore
0  gi|388476125|ref|YP_488308.1|   94.51     820   0.000000e+00    1573.0
1  gi|388476125|ref|YP_488308.1|   30.68     828  1.000000e-103     338.0
2  gi|388476125|ref|YP_488308.1|   30.90     466   3.000000e-44     164.0
```

Without `header=None` pandas would use the first BLAST hit as column names - and that hit would be missing from your data. (For `-outfmt 7`, which has `#` comment lines, add `comment="#"`.)

BED files

BED files are tab-separated with no header too; the first three columns are chromosome, start and end:

```
exons = pd.read_csv("rice_random_exons.bed", sep="\t", header=None,
                    names=["chrom", "start", "end"])
print(exons.head(3))
```

```
   chrom     start       end
0   Chr7  21408673  21408826
1   Chr9  16031526  16031938
2  Chr11   4762531   4762595
```

GFF3 files: skipping comment lines

A GFF3 file has 9 tab-separated columns and header lines starting with `#`. `comment="#"` skips them (and ignores anything after a `#` on a line, which is fine here - check that your file has no `#` inside its data):

```
gff_columns = ["seqid", "source", "type", "start", "end",
               "score", "strand", "phase", "attributes"]
gff = pd.read_csv("S_cerevisiae.gff3.gz", sep="\t", comment="#",
                  header=None, names=gff_columns)
print(gff.shape)
print(gff[["seqid", "type", "start", "end", "strand"]].head())
```

```
(23058, 9)
  seqid                            type  start     end strand
0  chrI                     chromosome      1  230218      .
1  chrI                       telomere      1     801      -
2  chrI                      X_element    337     801      -
3  chrI  X_element_combinatorial_repeat     63     336      -
4  chrI                telomeric_repeat      1      62      -
```

Some GFF3 files end with a `##FASTA` section holding the genome sequence; those lines would be read as bad rows. This yeast file has none; for files that do, cut the file at `##FASTA` first (e.g. `sed '/^##FASTA/,$d'`).

Useful `read_csv` options

Option	What it does	Example
`sep="\t"`	tab-separated (default is `,`)	BLAST, BED, GFF, VCF
`header=None`	the file has no header line	BLAST `-outfmt 6`, BED
`names=[...]`	column names to use	with `header=None`
`comment="#"`	skip `#` lines	GFF3, BLAST `-outfmt 7`

Option	What it does	Example
`usecols=[...]`	read only some columns (saves memory)	`usecols=["qseqid", "bitscore"]`
`index_col="gene"`	use a column as the row index	count tables
`dtype={"chrom": str}`	force a column's type	chromosome names `1`, `2` ...
`nrows=1000`	read only the first rows	peek at a huge file
`na_values=[...]`, `keep_default_na=False`	control what counts as missing	see Missing values

First look at a table

Always look before you analyze: how big is it, what are the columns, did the numbers come in as numbers?

```
print(gff.shape)          # rows, columns
print(gff.dtypes)         # the type of each column
```

```
(23058, 9)
seqid           str
source          str
type            str
start         int64
end           int64
score           str
strand          str
phase           str
attributes      str
dtype: object
```

`int64` and `float64` are whole and decimal numbers; `str` is text. If a column you expect to be numeric shows up as `str`, something in it isn't a number (a `-`, a comma, a stray header line) - find out why before going on.

`info()` shows the types plus how many values are **not missing** in each column:

```
species.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 143732 entries, 0 to 143731
Data columns (total 14 columns):
 #   Column               Non-Null Count   Dtype
---  ------               --------------   -----
 0   taxonid              143732 non-null  int64
 1   kingdom_name         143732 non-null  str
 2   phylum_name          143732 non-null  str
 3   class_name           143732 non-null  str
 4   order_name           143732 non-null  str
 5   family_name          143732 non-null  str
 6   genus_name           143732 non-null  str
 7   scientific_name      143732 non-null  str
 8   taxonomic_authority  143127 non-null  str
 9   infra_rank           2738 non-null    str
 10  infra_name           2738 non-null    str
 11  population           239 non-null     str
 12  category             143732 non-null  str
 13  main_common_name     56146 non-null   str
dtypes: int64(1), str(13)
memory usage: 29.7 MB
```

Only 56,146 of the 143,732 species have a common name, and `population` is almost always empty. `describe()` summarizes the numeric columns. Let's add a feature length column to the GFF table first (more on new columns below):

```
gff["length"] = gff["end"] - gff["start"] + 1
print(gff["length"].describe())
```

```
count    2.305800e+04
mean     1.759965e+03
std      2.183510e+04
min      1.000000e+00
25%      3.930000e+02
50%      9.420000e+02
75%      1.674000e+03
max      1.531933e+06
Name: length, dtype: float64
```

Large numbers are printed in scientific notation: `1.531933e+06` means 1.531933 x 1,000,000 = 1,531,933 bp. The largest "features" are the chromosomes themselves (the GFF has a `chromosome` line for each). Other quick looks: `df.head(10)`, `df.tail()`, `df.sample(5)` (random rows), `df.columns`, `len(df)`, and `df["col"].unique()` / `.nunique()`.

Selecting columns and rows

Columns

One column name in `[]` gives a Series; a **list** of names (note the double brackets) gives a smaller DataFrame:

```
print(species["category"].head(3))
print(species[["kingdom_name", "class_name", "scientific_name", "category"]].head(3))
```

```
0    LR/lc
1    LR/cd
2    LR/cd
Name: category, dtype: str
  kingdom_name     class_name      scientific_name category
0      PLANTAE  MAGNOLIOPSIDA    Eugenia oreophila    LR/lc
1      PLANTAE  MAGNOLIOPSIDA       Eugenia orites    LR/cd
2      PLANTAE  MAGNOLIOPSIDA  Eugenia pahangensis    LR/cd
```

Rows: boolean filters

The most common way to choose rows is a **condition**. A comparison on a column gives a Series of `True`/`False`, and putting that in `[]` keeps the `True` rows:

```
is_gene = gff["type"] == "gene"
print(is_gene.head(3))
genes = gff[is_gene]
print(len(gff), len(genes))
```

```
0    False
1    False
2    False
Name: type, dtype: bool
23058 6600
```

Combine conditions with `&` (and), `|` (or) and `~` (not). Each condition **must be in parentheses**, and you can't use the words `and`/`or` here:

```
long_plus = gff[(gff["type"] == "gene") & (gff["length"] > 5000) & (gff["strand"] == "+")]
print(len(long_plus), "genes over 5 kb on the + strand")

rnas = gff[gff["type"].isin(["tRNA_gene", "rRNA_gene", "snoRNA_gene"])]
print(rnas["type"].value_counts())

mito = gff[gff["seqid"] == "chrmt"]
print(mito["type"].value_counts().head(4))
```

```
44 genes over 5 kb on the + strand
type
tRNA_gene      299
snoRNA_gene     77
rRNA_gene       24
Name: count, dtype: int64
type
CDS               59
intron            32
noncoding_exon    28
gene              28
Name: count, dtype: int64
```

For text columns, `.str` gives you string methods for the whole column, such as `.str.contains()`, `.str.startswith()`, `.str.upper()`:

```
cats = species[species["genus_name"] == "Felis"]
print(cats[["scientific_name", "main_common_name", "category"]])
```

```
         scientific_name      main_common_name category
109001      Felis lybica  Afro-Asiatic Wildcat       LC
137435  Felis silvestris      European Wildcat       LC
```

```
tomatoes = species[species["scientific_name"].str.startswith("Solanum ")]
print(len(tomatoes), "Solanum species")
print(tomatoes["category"].value_counts().head(3).to_dict())
```

```
361 Solanum species
{'LC': 228, 'EN': 57, 'VU': 24}
```

`df.query()` is another way to write a filter, as a string that reads almost like SQL: `gff.query("type == 'gene' and length > 10000")`.

`.loc` and `.iloc`

- `.iloc[rows, cols]` selects by **position** (0, 1, 2 . . .), like a list.
- `.loc[rows, cols]` selects by **label** (index labels and column names), and also takes a condition for the rows.

```
print(gff.iloc[0])              # first row, as a Series
print(gff.iloc[0:3, 0:4])       # first 3 rows, first 4 columns
print(gff.loc[gff["type"] == "gene", ["seqid", "start", "end"]].head(3))
```

```
seqid                                          chrI
source                                          SGD
type                                    chromosome
start                                            1
end                                         230218
score                                            .
strand                                           .
phase                                            .
```

```
attributes    ID=chrI;dbxref=NCBI:NC_001133;Name=chrI
length                                        230218
Name: 0, dtype: object
  seqid source        type  start
0  chrI    SGD  chromosome      1
1  chrI    SGD    telomere      1
2  chrI    SGD   X_element    337
   seqid  start   end
5   chrI    335   649
8   chrI    538   792
12  chrI   1807  2169
```

`.loc` is most useful with a meaningful index. `set_index()` makes a column the index, and then you can look rows up by name, like a dictionary:

```
genes = gff[gff["type"] == "gene"].copy()
genes["gene_id"] = genes["attributes"].str.extract(r"ID=([^;]+)")
genes = genes.set_index("gene_id")
print(genes.loc["YFL039C", ["seqid", "start", "end", "strand", "length"]])
print(genes.loc[["YFL039C", "YAL001C"], ["seqid", "length"]])
```

```
seqid     chrVI
start     53260
end       54696
strand        -
length     1437
Name: YFL039C, dtype: object
         seqid  length
gene_id
YFL039C  chrVI    1437
YAL001C   chrI    3573
```

(`str.extract()` with the pattern `ID=([^;]+)` pulls out the text between `ID=` and the next `;` - regular expressions are the topic of Python V. The `.copy()` makes `genes` a separate table, not a view of part of `gff`, so adding columns to it doesn't trigger a warning.)

New columns

Assigning to a new column name adds a column. Arithmetic on columns works row by row, without a loop:

```
exons["length"] = exons["end"] - exons["start"]      # BED is 0-based: no +1
print(exons.head(3))
print(exons["length"].median())
```

```
   chrom     start       end  length
0   Chr7  21408673  21408826     153
1   Chr9  16031526  16031938     412
2  Chr11   4762531   4762595      64
175.5
```

GFF coordinates are 1-based and include the end, so a GFF feature length is `end - start + 1` (as we did above); BED coordinates are 0-based and exclude the end, so a BED length is `end - start`.

String methods make new columns from text. BLAST IDs like `gi|388476125|ref|YP_488308.1|` are long; split on `|` and keep the accession (item 3):

```
blast["qacc"] = blast["qseqid"].str.split("|").str[3]
blast["sacc"] = blast["sseqid"].str.split("|").str[3]
```

```
blast["qcov_len"] = blast["qend"] - blast["qstart"] + 1
print(blast[["qacc", "sacc", "pident", "qcov_len", "bitscore"]].head(3))
```

```
          qacc            sacc  pident  qcov_len  bitscore
0  YP_488308.1  YP_008250666.1   94.51       820    1573.0
1  YP_488308.1  YP_008251109.1   30.68       808     338.0
2  YP_488308.1  YP_008251012.1   30.90       458     164.0
```

Other useful column tools: `df["col"].round(1)`, `.abs()`, `np.log10(df["col"])` (with `import numpy as np`), `df.rename(columns={"old": "new"})`, `df.drop(columns=["col"])`, and `pd.cut()` to put numbers into bins:

```
bins = pd.cut(blast["pident"], bins=[0, 30, 50, 70, 90, 100])
print(bins.value_counts().sort_index())
```

```
pident
(0, 30]      10539
(30, 50]      6496
(50, 70]       502
(70, 90]      1366
(90, 100]     1646
Name: count, dtype: int64
```

Sorting and counting

```
genes = genes[["seqid", "start", "end", "strand", "length", "attributes"]]
print(genes.sort_values("length", ascending=False).head(4)[["seqid", "length"]])
print(genes.nlargest(3, "length")[["seqid", "length"]])     # the same, shorter
```

```
           seqid  length
gene_id
YLR106C   chrXII   14733
Q0045      chrmt   12884
YKR054C    chrXI   12279
YHR099W  chrVIII   11235
          seqid  length
gene_id
YLR106C  chrXII   14733
Q0045     chrmt   12884
YKR054C   chrXI   12279
```

Q0045 is the mitochondrial COX1 gene: its 12.9 kb includes several introns. Sort by more than one column with a list: `sort_values(["seqid", "start"])`.

`value_counts()` counts how often each value appears - the pandas version of `cut | sort | uniq -c | sort -rn`:

```
print(gff["type"].value_counts().head(8))
print(genes["strand"].value_counts())
print(blast["qacc"].nunique(), "queries with at least one hit")
```

```
type
CDS                    7058
gene                   6600
mRNA                   6600
noncoding_exon          484
long_terminal_repeat    383
intron                  377
ARS                     352
tRNA_gene               299
```

```
Name: count, dtype: int64
strand
+    3331
-    3269
Name: count, dtype: int64
3654 queries with at least one hit
```

Grouping and summarizing: `groupby`

`groupby` splits the table into groups by the values in one or more columns, computes something for each group, and puts the results back together (*split - apply - combine*). For example the mean gene length on each chromosome:

```
print(genes.groupby("seqid")["length"].mean().round(1).head())
```

```
seqid
chrI      1258.4
chrII     1337.7
chrIII    1204.9
chrIV     1342.9
chrIX     1338.3
Name: length, dtype: float64
```

`.agg()` computes several summaries at once, and **named aggregation** (`new_name=("column", "function")`) lets you name the result columns:

```
per_chrom = genes.groupby("seqid").agg(n_genes=("length", "size"),
                                       mean_length=("length", "mean"),
                                       longest=("length", "max"))
print(per_chrom.sort_values("n_genes", ascending=False).head())
```

```
        n_genes  mean_length  longest
seqid
chrIV       836  1342.948565     9807
chrXV       597  1319.053601     9240
chrVII      583  1345.017153     8019
chrXII      578  1372.517301    14733
chrXVI      511  1348.074364     7470
```

Common functions: `size` (number of rows), `count` (non-missing values), `sum`, `mean`, `median`, `min`, `max`, `std`, `nunique`, `first`. Group by several columns with a list; the result has one row per combination:

```
print(genes.groupby(["seqid", "strand"]).size().head(4))
```

```
seqid  strand
chrI   +          60
       -          57
chrII  +         211
       -         245
dtype: int64
```

How many BLAST hits (HSPs) does each *E. coli* protein have? Most have one or two, but members of big families (transporters, regulators) hit many *Salmonella* proteins:

```
hits_per_query = blast.groupby("qacc").size()
print(hits_per_query.describe())
print(hits_per_query.sort_values(ascending=False).head(3))
```

```
count    3654.000000
mean        5.623700
```

```
std        13.697262
min         1.000000
25%         1.000000
50%         2.000000
75%         4.000000
max       141.000000
dtype: float64
qacc
YP_489067.1    141
YP_490420.1    139
YP_491948.1    138
dtype: int64
```

Joining tables: `merge`

The BLAST table only has IDs. The protein names are in the FASTA headers of `E_coli_K12.pep.gz`. Let's make a second table from the FASTA file with Biopython (Packages lecture), then **join** the two tables on the query ID - just like `JOIN` in SQL or `merge()` in R:

```
import gzip
from Bio import SeqIO

rows = []
with gzip.open("E_coli_K12.pep.gz", "rt") as fh:
    for record in SeqIO.parse(fh, "fasta"):
        # description: "gi|...|ref|YP_488307.1| thr operon leader peptide [Escherichia ...]"
        name = record.description.removeprefix(record.id).split(" [")[0].strip()
        rows.append({"qseqid": record.id, "name": name, "qlen": len(record)})
proteins = pd.DataFrame(rows)       # a list of dictionaries -> one row each
print(proteins.shape)
print(proteins[["name", "qlen"]].head(3))
```

```
(4213, 3)
                                              name  qlen
0                          thr operon leader peptide    21
1  fused aspartokinase I and homoserine dehydroge...   820
2                                homoserine kinase   310
```

`merge()` matches rows that have the same value in the `on` column(s):

```
hits = blast.merge(proteins, on="qseqid")
print(blast.shape, hits.shape)
print(hits[["qacc", "pident", "name"]].tail(3))
```

```
(20549, 15) (20549, 17)
              qacc  pident                                              name
20546  YP_492531.1   21.70  DNA-binding response regulator in two-componen...
20547  YP_492533.1   82.46                            rRNA methyltransferase
20548  YP_492533.1   36.60                            rRNA methyltransferase
```

The `how=` option decides what happens to rows without a partner:

`how=`	Keeps	Like SQL
`"inner"` (default)	only keys found in both tables	`JOIN`
`"left"`	every row of the left table; missing values where there is no match	`LEFT JOIN`

how=	Keeps	Like SQL
"right"	every row of the right table	RIGHT JOIN
"outer"	every row of both	FULL JOIN

A left join starting from **all** proteins tells us which *E. coli* proteins have **no** hit in *Salmonella* - their BLAST columns are missing (NaN):

```
all_proteins = proteins.merge(blast, on="qseqid", how="left")
no_hit = all_proteins[all_proteins["sseqid"].isna()]
print(len(proteins), "proteins,", len(no_hit), "without a hit")
print(no_hit["name"].value_counts().head(6))
```

```
4213 proteins, 559 without a hit
name
inner membrane protein                          32
DNA-binding transcriptional regulator           19
IS5 transposase and trans-activator             11
general secretory pathway component, cryptic     8
IS5 element protein                              7
IS1 repressor protein InsA                       6
Name: count, dtype: int64
```

Many of the *E. coli*-only proteins come from mobile DNA (IS elements, transposases). If the key columns have different names, use `left_on="qseqid", right_on="id"`. And check the row counts before and after a merge: if a key appears several times in *both* tables you get every combination, and the table can grow unexpectedly.

Missing values and the "NA" trap

Missing values in pandas are shown as `NaN` (Not a Number; for text columns you may also see `None` or `<NA>`). By default `read_csv()` treats empty fields **and** a list of strings including `NA`, `N/A`, `NaN`, `NULL`, `null` and `None` as missing. That is convenient until your data contains the *real* text "NA". The influenza neuraminidase gene is called NA! Save this small table as `flu_segments.csv`:

```
cat > flu_segments.csv << 'EOF'
segment,gene,length,protein
1,PB2,2341,polymerase PB2
2,PB1,2341,polymerase PB1
3,PA,2233,polymerase PA
4,HA,1778,hemagglutinin
5,NP,1565,nucleoprotein
6,NA,1413,neuraminidase
7,M,1027,
8,NS,890,
EOF
```

```
flu = pd.read_csv("flu_segments.csv")
print(flu)
print(flu.isna().sum())       # missing values per column
```

```
   segment gene  length         protein
0        1  PB2    2341  polymerase PB2
1        2  PB1    2341  polymerase PB1
2        3   PA    2233   polymerase PA
3        4   HA    1778   hemagglutinin
4        5   NP    1565   nucleoprotein
5        6  NaN    1413   neuraminidase
```

```
6        7    M    1027             NaN
7        8   NS     890             NaN
segment    0
gene       1
length     0
protein    2
dtype: int64
```

The gene name `NA` became `NaN`. The same can happen to Namibia's country code (`NA`), to "not applicable" codes, and to any column where `NA` or `null` is a real value. The fix is to say exactly what counts as missing:

```
flu = pd.read_csv("flu_segments.csv", keep_default_na=False, na_values=[""])
print(flu[["gene", "protein"]].tail(3))
```

```
  gene        protein
5   NA  neuraminidase
6    M            NaN
7   NS            NaN
```

Now only empty fields are missing. Working with missing values:

```
print(flu[flu["protein"].isna()])                  # rows with a missing protein
print(flu["protein"].fillna("unknown").tolist())   # replace missing values
print(len(flu.dropna()), "rows with no missing values")
```

```
   segment gene  length protein
6        7    M    1027     NaN
7        8   NS     890     NaN
['polymerase PB2', 'polymerase PB1', 'polymerase PA', 'hemagglutinin', 'nucleoprotein', 'neuraminidase',
6 rows with no missing values
```

`isna()`/`notna()` test for missing values, `fillna(value)` replaces them, and `dropna()` removes rows that have any (or use `dropna(subset=["col"])`). Most summaries skip missing values: `mean()` is the mean of the values that are present, and `count()` counts only non-missing values while `size` counts all rows. Never test `x == np.nan` - it is always `False`; use `isna()`.

Reshaping tables

Two-way counts: `crosstab`

`pd.crosstab()` counts combinations of two columns. Here, species per kingdom and Red List category:

```
table = pd.crosstab(species["kingdom_name"], species["category"])
print(table[["CR", "EN", "VU", "NT", "LC", "DD"]])
```

```
category        CR     EN    VU    NT     LC     DD
kingdom_name
ANIMALIA      3136   5090  5532  4350  46491  14711
CHROMISTA        4      1     1     0      0      9
FUNGI           35    101   154    61    211     65
PLANTAE       5401  10314  9580  3370  28545   5112
```

Wide and long: `pivot_table`, `melt` and `pivot`

The crosstab above is a **wide** table: one row per kingdom, one column per category. Many tools (seaborn plots, SQL, R's ggplot2) prefer **long** ("tidy") data: one row per observation, here one row per kingdom-category pair. `melt()` goes from wide to long, and `pivot()` goes back:

```
wide = table[["CR", "EN", "VU"]].reset_index()      # kingdom_name becomes a column
print(wide)
```

```
long = wide.melt(id_vars="kingdom_name", var_name="category", value_name="n_species")
print(long.head(5))
back = long.pivot(index="kingdom_name", columns="category", values="n_species")
print(back)
```

```
category kingdom_name    CR     EN    VU
0             ANIMALIA  3136   5090  5532
1            CHROMISTA     4      1     1
2                FUNGI    35    101   154
3              PLANTAE  5401  10314  9580
  kingdom_name category  n_species
0     ANIMALIA       CR       3136
1    CHROMISTA       CR          4
2        FUNGI       CR         35
3      PLANTAE       CR       5401
4     ANIMALIA       EN       5090
category        CR     EN    VU
kingdom_name
ANIMALIA      3136   5090  5532
CHROMISTA        4      1     1
FUNGI           35    101   154
PLANTAE       5401  10314  9580
```

`pivot_table()` is `groupby` + `pivot` in one step, for summaries other than counts - e.g. the median gene length per chromosome and strand:

```
print(genes.pivot_table(index="seqid", columns="strand", values="length",
                        aggfunc="median").head(4))
```

```
strand       +       -
seqid
chrI     849.0   921.0
chrII    990.0  1140.0
chrIII  1008.0   868.0
chrIV   1134.0  1066.5
```

Writing tables

```
import os

best_cols = ["qacc", "sacc", "pident", "length", "evalue", "bitscore"]
blast[best_cols].to_csv("blast_hits.csv", index=False)                 # CSV
blast[best_cols].to_csv("blast_hits.tsv", sep="\t", index=False)       # TSV
blast[best_cols].to_csv("blast_hits.tsv.gz", sep="\t", index=False)    # compressed TSV
blast[best_cols].to_parquet("blast_hits.parquet")                      # Parquet

for f in ["blast_hits.csv", "blast_hits.tsv.gz", "blast_hits.parquet"]:
    print(f, os.path.getsize(f), "bytes")
print(pd.read_parquet("blast_hits.parquet").dtypes)
```

```
blast_hits.csv 1028129 bytes
blast_hits.tsv.gz 255756 bytes
blast_hits.parquet 236948 bytes
qacc            str
sacc            str
pident      float64
length        int64
```

```
evalue      float64
bitscore    float64
dtype: object
```

- `index=False` leaves out the row numbers; without it you get an unnamed extra first column. (Keep the index when it means something, e.g. gene IDs after `set_index()`.)
- A `.gz` file name makes pandas compress the output.
- **Parquet** files are compressed, fast to read, and store the column types (a text file doesn't: when you read a CSV pandas has to guess again). They are great for big intermediate tables you will read again from Python, R or DuckDB, but you can't look at them with `less`. The DuckDB lecture explains CSV vs Parquet in more detail.

Worked examples

1. Yeast genome annotation summary

How many genes and tRNA genes are on each yeast chromosome, and how dense are genes? The chromosome sizes are in the GFF itself (the `chromosome` lines), so we make two small tables and join them.

```python
#!/usr/bin/env python3
# Summarize the yeast genome annotation: genes per chromosome and gene density
import pandas as pd

gff_columns = ["seqid", "source", "type", "start", "end",
               "score", "strand", "phase", "attributes"]
gff = pd.read_csv("S_cerevisiae.gff3.gz", sep="\t", comment="#",
                  header=None, names=gff_columns)
gff["length"] = gff["end"] - gff["start"] + 1

# 1. chromosome sizes come from the 'chromosome' lines
chroms = gff.loc[gff["type"] == "chromosome", ["seqid", "length"]]
chroms = chroms.rename(columns={"length": "chrom_bp"})

# 2. count genes and tRNA genes on each chromosome
features = gff[gff["type"].isin(["gene", "tRNA_gene"])]
counts = pd.crosstab(features["seqid"], features["type"]).reset_index()

# 3. join the two tables and compute genes per 100 kb
summary = chroms.merge(counts, on="seqid")
summary["genes_per_100kb"] = (summary["gene"] / summary["chrom_bp"] * 1e5).round(1)
summary = summary.sort_values("chrom_bp", ascending=False)
print(summary.to_string(index=False))
summary.to_csv("yeast_chrom_summary.tsv", sep="\t", index=False)
```

```
  seqid  chrom_bp  gene  tRNA_gene  genes_per_100kb
  chrIV   1531933   836         28             54.6
  chrXV   1091291   597         20             54.7
 chrVII   1090940   583         36             53.4
 chrXII   1078177   578         21             53.6
 chrXVI    948066   511         17             53.9
chrXIII    924431   505         21             54.6
  chrII    813184   456         13             56.1
 chrXIV    784333   435         14             55.5
   chrX    745751   398         24             53.4
  chrXI    666816   348         16             52.2
   chrV    576874   323         20             56.0
```

```
chrVIII    562643   321         11             57.1
  chrIX    439888   241         10             54.8
 chrIII    316620   184         10             58.1
  chrVI    270161   139         10             51.5
   chrI    230218   117          4             50.8
  chrmt     85779    28         24             32.6
```

The nuclear chromosomes all have about 50-58 genes per 100 kb - yeast genes are tightly packed - while the mitochondrial genome (`chrmt`) has far fewer protein genes but 24 tRNA genes. `to_string(index=False)` prints the whole table without the index.

2. The best BLAST hit for each query

A very common task: keep only the top-scoring hit for each query. Sort so that the best hit of each query comes first, then keep the first row of each query with `drop_duplicates()`:

```
#!/usr/bin/env python3
# Best Salmonella hit for each E. coli protein
import pandas as pd

blast_columns = ["qseqid", "sseqid", "pident", "length", "mismatch", "gapopen",
                 "qstart", "qend", "sstart", "send", "evalue", "bitscore"]
blast = pd.read_csv("Ecoli-vs-Senterica.BLASTP.tab.gz", sep="\t",
                    header=None, names=blast_columns)

best = (blast.sort_values(["qseqid", "bitscore"], ascending=[True, False])
             .drop_duplicates(subset="qseqid", keep="first"))
print(len(blast), "hits,", len(best), "best hits")
print(best[["qseqid", "pident", "length", "bitscore"]].head(3))

# the same with idxmax: the row label of the highest bitscore in each group
best2 = blast.loc[blast.groupby("qseqid")["bitscore"].idxmax()]
print("same best hits:", (best["sseqid"].values == best2["sseqid"].values).all())

print(best["pident"].describe().round(1))
print((best["pident"] >= 90).sum(), "E. coli proteins have a best hit >= 90% identical")
```

```
20549 hits, 3654 best hits
                          qseqid  pident  length  bitscore
0  gi|388476125|ref|YP_488308.1|   94.51     820    1573.0
4  gi|388476126|ref|YP_488309.1|   93.51     308     599.0
5  gi|388476127|ref|YP_488310.1|   93.22     428     802.0
same best hits: True
count    3654.0
mean       80.1
std        21.0
min        19.0
25%        77.3
50%        88.7
75%        94.0
max       100.0
Name: pident, dtype: float64
1642 E. coli proteins have a best hit >= 90% identical
```

(Wrapping a chain of methods in parentheses lets you put one step per line.) Two details matter here:

- **Ties.** 14 queries have two or more hits with exactly the same best bitscore. Sorting on `["qseqid", "bitscore"]` keeps the tied hits in their original file order, so both methods keep the first one. Sorting on

`bitscore` alone uses a faster, unstable sort and can pick a different one of the tied hits. If ties matter, add a tie-breaker column such as `pident`.

- **Look beyond the E-value.** Add the query length from the `proteins` table (the merge section) to compute how much of each query the best hit covers: `100 * (qend - qstart + 1) / qlen`. A short, very similar piece of a long protein is not the same thing as a full-length ortholog.

The DuckDB lecture does the same best-hit query in SQL with `arg_max()` and `QUALIFY row_number() ...`.

3. Threatened species

Which groups have the highest proportion of threatened species? The IUCN counts `CR`, `EN` and `VU` as "threatened". A `True`/`False` column can be summed (True = 1), so its sum is a count and its mean is a proportion:

```
#!/usr/bin/env python3
# Proportion of threatened species per class, and families with most CR species
import pandas as pd

species = pd.read_csv("threatened-species.csv.gz")
species["threatened"] = species["category"].isin(["CR", "EN", "VU"])

by_class = species.groupby("class_name").agg(assessed=("taxonid", "size"),
                                             threatened=("threatened", "sum"))
by_class["pct_threatened"] = 100 * by_class["threatened"] / by_class["assessed"]
by_class["pct_threatened"] = by_class["pct_threatened"].round(1)
big = by_class[by_class["assessed"] >= 1000]          # skip tiny groups
print(big.sort_values("pct_threatened", ascending=False).head(8))

cr_families = (species[species["category"] == "CR"]
               .groupby(["kingdom_name", "family_name"]).size()
               .sort_values(ascending=False).head(5))
print(cr_families)
```

```
                assessed  threatened  pct_threatened
class_name
MAGNOLIOPSIDA      51934       20851            40.1
LILIOPSIDA          9133        3493            38.2
AMPHIBIA            7258        2471            34.0
CHONDRICHTHYES      1256         406            32.3
GASTROPODA          5752        1440            25.0
MAMMALIA            2697         668            24.8
REPTILIA            9835        1618            16.5
INSECTA            12131        1940            16.0
kingdom_name  family_name
PLANTAE       RUBIACEAE      303
              FABACEAE       269
              LAURACEAE      266
              ORCHIDACEAE    261
              MYRTACEAE      251
dtype: int64
```

Flowering plants (Magnoliopsida, Liliopsida), amphibians and sharks and rays (Chondrichthyes) have the highest proportions among the well-sampled classes. Keep in mind what this table is: species that have been *assessed*, and groups of concern are more likely to be assessed. The Plotting lecture plots these counts.

pandas, a loop, SQL or awk?

The same question can be answered many ways. Counting genes per chromosome:

```
zcat S_cerevisiae.gff3.gz | awk -F'\t' '$3 == "gene" {n[$1]++} END {print n["chrIV"]}'
```

```
836
```

```
# plain Python: a loop and a dictionary
import gzip
counts = {}
with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        cols = line.rstrip("\n").split("\t")
        if not line.startswith("#") and len(cols) == 9 and cols[2] == "gene":
            counts[cols[0]] = counts.get(cols[0], 0) + 1
print(counts["chrIV"])

# pandas
print(gff.loc[gff["type"] == "gene", "seqid"].value_counts()["chrIV"])
```

```
836
836
```

```
-- DuckDB SQL (see the SQL_DuckDB lecture); column0, column1 ... are the default names
SELECT column0 AS seqid, count(*) AS n_genes
FROM read_csv('S_cerevisiae.gff3.gz', delim = '\t', header = false, comment = '#')
WHERE column2 = 'gene' AND column0 = 'chrIV'
GROUP BY seqid;
```

All three give 836.

Use	When
`awk`, `cut`, `sort`, `uniq`, `grep`	quick one-off questions on the command line; simple filters and counts on files of any size; pipelines in job scripts
a plain Python loop	the file is not really a table (FASTA, GenBank, nested records), each line needs complicated logic, or the file is far too big for memory and you only need one pass
pandas	table questions with several steps: filter, new columns, group, join two tables, reshape, then plot or feed a statistics package; the table fits in memory (up to a few GB)
DuckDB / SQL	the same table operations when the files are too big for pandas (DuckDB streams from disk), when you want to query many CSV/Parquet files at once, or when you already know SQL; it can also return a pandas DataFrame

A rough guide to memory: a pandas DataFrame needs a few times the size of the uncompressed text file. `df.info(memory_usage="deep")` reports the real number.

pandas and R's dplyr

If you know R (or are learning it in the R plotting lecture), the same operations have different names:

Task	pandas	dplyr / R
read a CSV	`pd.read_csv("f.csv")`	`read_csv("f.csv")`
keep rows	`df[df["len"] > 100]` or `df.query("len > 100")`	`filter(df, len > 100)`
keep columns	`df[["a", "b"]]`	`select(df, a, b)`
new column	`df["kb"] = df["len"] / 1000`	`mutate(df, kb = len / 1000)`

Task	pandas	dplyr / R
sort	`df.sort_values("len", ascending=False)`	`arrange(df, desc(len))`
count values	`df["type"].value_counts()`	`count(df, type)`
group summaries	`df.groupby("chr").agg(n=("len", "size"))`	`df %>% group_by(chr) %>% summarize(n = n())`
join	`a.merge(b, on="id", how="left")`	`left_join(a, b, by = "id")`
wide to long	`df.melt(...)`	`pivot_longer(...)`
long to wide	`df.pivot(...)`	`pivot_wider(...)`
missing value	`NaN`, `df["x"].isna()`	`NA`, `is.na(x)`
rows and columns count from	0	1

Common mistakes

- **Forgetting `header=None`** on BLAST or BED files: the first data line becomes the column names and is lost from the data.
- **Forgetting `sep="\t"`**: a TSV read as CSV has one giant column. Check `df.shape`.
- **The "NA" trap**: gene names, country codes or genotypes like `NA` read as missing. Use `keep_default_na=False, na_values=[""]`.
- **`and`/`or` in filters**: `df[(a > 1) and (b < 2)]` raises `ValueError: The truth value of a Series is ambiguous`. Use `&`, `|` and parentheses: `df[(df["a"] > 1) & (df["b"] < 2)]`.
- **Missing parentheses with `&`**: `df["a"] > 1 & df["b"] < 2` is evaluated in the wrong order. Put each condition in `( )`.
- **Numbers read as text**: a column with one bad value (`"-"`, `"1,234"`) comes in as `str`, and `"100" > "99"` is `False` for text. Check `dtypes`; convert with `pd.to_numeric(df["col"], errors="coerce")` (bad values become `NaN`).
- **Off-by-one lengths**: GFF length is `end - start + 1`, BED length is `end - start`.
- **Writing the index by accident**: `to_csv("out.csv")` adds an unnamed first column. Use `index=False`.
- **Looping over rows** (`for i, row in df.iterrows():`) to compute a new column. It works but is slow and long; use column arithmetic (`df["a"] - df["b"]`) or `.str` methods instead.
- **Chained assignment**: `df[df["type"] == "gene"]["length"] = 0` changes a temporary copy, not `df` (pandas 3 never changes `df` this way; older versions print a `SettingWithCopyWarning`). Use `.loc`: `df.loc[df["type"] == "gene", "length"] = 0`.
- **Merges that multiply rows**: joining on a key that is repeated in both tables. Check `len()` before and after, and `df["key"].duplicated().sum()`.
- **Ties when picking a best hit**: decide which one you want and sort on a tie-breaker.

Quick reference

Task	Code
import	`import pandas as pd`
read CSV / TSV	`pd.read_csv(f)`, `pd.read_csv(f, sep="\t")`
no header	`pd.read_csv(f, sep="\t", header=None, names=cols)`
skip comments	`pd.read_csv(f, sep="\t", comment="#")`
read Parquet	`pd.read_parquet(f)`
size, types	`df.shape`, `df.dtypes`, `df.info()`, `df.describe()`
look	`df.head()`, `df.tail()`, `df.sample(5)`, `df.columns`
one column / several	`df["a"]`, `df[["a", "b"]]`

Task	Code
filter rows	`df[df["a"] > 5]`, `df[(cond1) & (cond2)]`, `df[df["a"].isin(list)]`
text filter	`df[df["name"].str.contains("kinase")]`
by position / label	`df.iloc[0:5, 0:2]`, `df.loc[rows, ["a", "b"]]`
new column	`df["len"] = df["end"] - df["start"] + 1`
split a text column	`df["attributes"].str.split(";").str[0]`
rename / drop	`df.rename(columns={"a": "b"})`, `df.drop(columns=["a"])`
sort	`df.sort_values("a", ascending=False)`, `df.nlargest(10, "a")`
count values	`df["a"].value_counts()`, `df["a"].nunique()`
group	`df.groupby("g")["a"].mean()`, `df.groupby("g").agg(n=("a", "size"))`
two-way counts	`pd.crosstab(df["a"], df["b"])`
join	`a.merge(b, on="key", how="left")`
missing values	`df.isna().sum()`, `df.dropna()`, `df.fillna(0)`
reshape	`df.melt(id_vars=...)`, `df.pivot(index=, columns=, values=)`, `df.pivot_table(...)`
best row per group	`df.sort_values(["g", "score"], ascending=[True, False]).drop_duplicates("g")`
write	`df.to_csv(f, sep="\t", index=False)`, `df.to_parquet(f)`

Exercises

1. Read `rice_random_exons.bed`. How many exons are on each chromosome? What are the mean, median, shortest and longest exon lengths? How many exons are shorter than 50 bp?
2. From the yeast GFF, make a table of **CDS** features and compute their lengths. Which gene has the most CDS pieces (exons)? (Hint: the `Parent` attribute names the mRNA, e.g. `Parent=YAL069W_mRNA` - use `str.extract(r"Parent=([^;]+)_mRNA")` and `value_counts()`.) How many yeast genes have more than one CDS piece, i.e. contain an intron?
3. Using `threatened-species.csv.gz`: how many species have no common name? What percentage is that for plants and for animals? Which 5 orders of mammals (`class_name == "MAMMALIA"`) have the most threatened (`CR`, `EN`, `VU`) species?
4. From the BLAST table, find the E. coli proteins whose best *Salmonella* hit is less than 50% identical. Merge in the protein names and lengths (from `E_coli_K12.pep.gz`) and compute the query coverage of the best hit. Write the table, sorted by percent identity, to a TSV file with a header.
5. Is there a relationship between the percent identity of the best hit and the protein length? Put the proteins into length bins with `pd.cut()` (e.g. 0-100, 100-300, 300-600, 600+ aa) and compute the median best-hit identity per bin.
6. Reshape: make a table with one row per yeast chromosome and one column per strand (`+`, `-`) holding the number of genes, using `pivot_table` or `crosstab`. Then `melt` it back into a long table and save both as CSV. Which chromosome has the most uneven strand split?
7. `Orthogroups.csv` from the course data is an OrthoFinder result for three cyanobacteria: one row per orthogroup, one column per species, and the genes of that species in the group as a comma-separated list. Note that the file is actually tab-separated, and the first column (the orthogroup ID) has no name in the header. Read it, count the number of genes per species in each orthogroup (hint: `.str.split(", ").str.len()`, and empty cells are 0), and find how many orthogroups have exactly one gene in every species (single-copy orthologs).
8. Challenge: write a command-line script (with `argparse`, from the Packages lecture) that reads any BLAST `-outfmt 6` file and writes the best hit per query, with options `--min-pident`, `--min-length` and

`--max-evalue`. Test it on the E. coli vs Salmonella table and compare the number of queries with a best hit at different thresholds. Then answer the same question with DuckDB from the SQL lecture and check that you get the same numbers.

Next

- Python V: Regular expressions - patterns in text and sequences, including `str.extract()` and `str.contains()` in pandas.
- Plotting - figures from DataFrames with matplotlib and seaborn.
- Tabular data with DuckDB - the same table operations in SQL, and CSV vs Parquet.
- The pandas documentation, especially "10 minutes to pandas" and the "Comparison with R / SQL / spreadsheets" pages in the user guide.