Python Workshop: Practice with Real Data

This workshop is a set of practice problems that use everything from Python I, Python II and Python III: strings, lists, loops, reading files, the `csv` module, dictionaries, sets and functions. The problems go from easy to harder. Each one has hints and a complete, tested solution.

How to use this workshop

- Try each problem yourself **before** you look at the solution. Write a script (for example `problem1.py`) in a folder on the cluster and run it with `python3 problem1.py`.
- Start small: first print the first few lines or fields to make sure you are reading the file correctly, then add the counting or calculation.
- Check your answer another way when you can, for example with a UNIX one-liner (`wc -l`, `cut`, `sort | uniq -c`, `awk`).
- There are many correct answers. If your numbers match the solution, compare your code with it anyway and look for ideas.

What you'll practice

Problem	Data	Skills
1. Exon lengths	BED	reading a tab-delimited file, `int()`, totals, min/max/mean
2. Exons per chromosome	BED	counting with a dict, sorting with `key=`, histogram bins
3. Threatened species	CSV	`csv` module, quoted fields, `Counter`, sets per key
4. FASTA statistics	FASTA	a FASTA reader function, GC content, codon counts
5. Translate ORFs	FASTA + codon table	a lookup dict, functions, checking results
6. Mini-project	GFF + FASTA	joining two files by gene ID, a summary table

Getting the data

Download the files once, in the shell, into the folder where you will work (on the cluster, somewhere in your home or bigdata folder):

```
CLASS=https://raw.githubusercontent.com/biodataprog/GEN220/master/data
DATA=https://github.com/biodataprog/GEN220_data/raw/main
curl -L -O $CLASS/rice_random_exons.bed
curl -L -O $CLASS/codon_table.txt
curl -L -O $DATA/tabular/threatened-species.csv.gz
curl -L -O $DATA/genome/S_cerevisiae.ORFs.fasta.gz
curl -L -O $DATA/genome/S_cerevisiae.gff3.gz
```

If you prefer to download from inside Python, use `urllib.request.urlretrieve()` from the standard library (rather than running `curl` through `os.system()`):

```
import os
import urllib.request

base = "https://raw.githubusercontent.com/biodataprog/GEN220/master/data/"
url = base + "rice_random_exons.bed"
filename = os.path.basename(url)          # "rice_random_exons.bed"
```

```
if not os.path.exists(filename):          # only download it once
    urllib.request.urlretrieve(url, filename)
```

Leave the `.gz` files compressed and read them with `gzip.open(filename, "rt")` (the `"rt"` means read as text).

Problem 1 (easy): exon lengths from a BED file

`rice_random_exons.bed` has 1,000 rice exons in BED format: tab-separated columns for chromosome, start and end.

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

Write a script that:

1. opens the file and loops over the lines,
2. prints the length of the first 5 exons,
3. reports the number of exons, the total length (bp and kb), and the shortest, longest and mean exon length.

Hints

- `line.rstrip("\n").split("\t")` gives a list of columns. The columns are strings: convert with `int()`.
- BED coordinates start at 0 and the end is not included ("0-based, half-open"), so the length is simply `end - start`. (GFF coordinates start at 1 and include the end, so there the length is `end - start + 1`.)
- Collect the lengths in a list; then `len()`, `sum()`, `min()` and `max()` do the rest.

Check with UNIX: `awk '{total += $3 - $2} END {print NR, total}' rice_random_exons.bed`

Solution

```
bedfile = "rice_random_exons.bed"

lengths = []
with open(bedfile) as fh:
    for line in fh:
        cols = line.rstrip("\n").split("\t")
        chrom, start, end = cols[0], int(cols[1]), int(cols[2])
        length = end - start
        if len(lengths) < 5:
            print(chrom, start, end, length)
        lengths.append(length)

total = sum(lengths)
print("exons:", len(lengths))
print(f"total length: {total} bp ({total / 1000:.1f} kb)")
print("shortest:", min(lengths), "longest:", max(lengths))
print(f"mean length: {total / len(lengths):.1f} bp")
```

```
Chr7 21408673 21408826 153
Chr9 16031526 16031938 412
Chr11 4762531 4762595 64
Chr8 54040 54193 153
Chr10 19815475 19815747 272
exons: 1000
total length: 369855 bp (369.9 kb)
shortest: 13 longest: 5818
mean length: 369.9 bp
```

Problem 2: exons per chromosome and a length histogram

Using the same BED file:

1. Count the exons on each chromosome and print them in chromosome order (`Chr1`, `Chr2`, ... `Chr12`, **not** `Chr1`, `Chr10`, `Chr11`, `Chr12`, `Chr2`). Look carefully at the chromosome names: is there one that is not a number?
2. Print the chromosomes again, ordered from most to fewest exons.
3. Make a histogram of exon lengths in 500 bp bins (0-499, 500-999, ...) and print the count for every bin, including empty ones, up to the largest bin.

Hints

- Count with `counts[chrom] = counts.get(chrom, 0) + 1`.
- Sorting chromosome names as text puts `Chr10` before `Chr2`. Write a small function that turns `"Chr10"` into the number `10` and use it as the `key=` for `sorted()`. The file also has one exon on `ChrSy` (sequence not assigned to a chromosome), and `int("Sy")` is an error, so give names without a number a large value to sort them last. Real data always has a surprise like this.
- To sort by count: `sorted(counts.items(), key=lambda pair: pair[1], reverse=True)`.
- The bin for a length is `length // 500 * 500` (`//` is integer division: `1257 // 500` is 2, so the bin starts at 1000). Use `range(0, largest_bin + 1, 500)` and `.get(bin_start, 0)` to print empty bins as 0.

Check with UNIX: `cut -f1 rice_random_exons.bed | sort | uniq -c | sort -k1,1nr`

Solution

```
bedfile = "rice_random_exons.bed"
bin_size = 500


def chrom_number(name):
    """Turn 'Chr10' into 10 for sorting; names without a number sort last."""
    number = name.replace("Chr", "")
    if number.isdigit():
        return int(number)
    return 1000


exons_per_chrom = {}
length_bins = {}
with open(bedfile) as fh:
    for line in fh:
        chrom, start, end = line.rstrip("\n").split("\t")
        length = int(end) - int(start)
        exons_per_chrom[chrom] = exons_per_chrom.get(chrom, 0) + 1
        bin_start = length // bin_size * bin_size
        length_bins[bin_start] = length_bins.get(bin_start, 0) + 1

print("Exons per chromosome, in chromosome order:")
for chrom in sorted(exons_per_chrom, key=chrom_number):
    print(f"{chrom}\t{exons_per_chrom[chrom]}")

print("Chromosomes with the most exons:")
ranked = sorted(exons_per_chrom.items(), key=lambda pair: pair[1], reverse=True)
for chrom, n in ranked[:3]:
    print(f"{chrom}\t{n}")

print("Exon length histogram:")
for bin_start in range(0, max(length_bins) + 1, bin_size):
```

```
    n = length_bins.get(bin_start, 0)
    bar = "#" * (n // 20)              # a text bar: one # per 20 exons
    print(f"{bin_start}-{bin_start + bin_size - 1}\t{n}\t{bar}")
```

```
Exons per chromosome, in chromosome order:
Chr1    146
Chr2    123
Chr3    122
Chr4    92
Chr5    77
Chr6    66
Chr7    71
Chr8    70
Chr9    50
Chr10   55
Chr11   67
Chr12   60
ChrSy   1
Chromosomes with the most exons:
Chr1    146
Chr2    123
Chr3    122
Exon length histogram:
0-499   795 ########################################
500-999 127 ######
1000-1499   37  #
1500-1999   20  #
2000-2499   7
2500-2999   5
3000-3499   3
3500-3999   5
4000-4499   0
4500-4999   0
5000-5499   0
5500-5999   1
```

Problem 3: threatened species per genus (CSV)

`threatened-species.csv.gz` is a table of species from the IUCN Red List. The first line is a header with 14 column names: `taxonid`, `kingdom_name`, `phylum_name`, `class_name`, `order_name`, `family_name`, `genus_name`, `scientific_name`, `taxonomic_authority`, `infra_rank`, `infra_name`, `population`, `category` and `main_common_name`. A typical row (note the quoted authority, which contains a comma):

```
54532,ANIMALIA,CHORDATA,AMPHIBIA,ANURA,BUFONIDAE,Atelopus,Atelopus nanay,"Coloma, 2002",,,,CR,
```

1. Show that `line.split(",")` does **not** work on this file: count how many lines do not split into exactly 14 fields.
2. Using the `csv` module, count the rows in each kingdom.
3. List the 10 genera with the most **distinct species names** (`scientific_name`), with their counts.
4. For fungi only, count species in each Red List `category`, and list the fungal genera with the most species in the threatened categories: `CR` (critically endangered), `EN` (endangered) and `VU` (vulnerable).

Hints

- Some fields contain commas inside quotes, for example the authority `"(Linnaeus, 1758)"`. `split(",")` breaks these apart; the `csv` module understands quoting.
- `csv.DictReader(fh)` reads the header for you and gives each row as a dictionary, so you can write `row["genus_name"]` instead of remembering that the genus is column 6.

- The same species name can appear on several rows (subspecies, regional populations). To count distinct names per genus, keep a **set** of names for each genus: `defaultdict(set)` and `.add()`.
- `{"CR", "EN", "VU"}` is a set; `if row["category"] in threatened:` tests membership.

Solution (also available as workshop_species.py)

```
#!/usr/bin/env python3
"""Workshop problem 3: count species per kingdom, genus and category."""
import csv
import gzip
from collections import Counter, defaultdict

datafile = "threatened-species.csv.gz"
threatened = {"CR", "EN", "VU"}

# 1. why we need the csv module: count lines that do not split into 14 fields
bad_lines = 0
with gzip.open(datafile, "rt") as fh:
    for line in fh:
        if len(line.rstrip("\n").split(",")) != 14:
            bad_lines += 1
print("lines that split(',') gets wrong:", bad_lines)

kingdoms = Counter()
species_in_genus = defaultdict(set)      # genus -> set of species names
fungal_categories = Counter()
threatened_fungi = Counter()             # genus -> number of threatened rows

with gzip.open(datafile, "rt", newline="") as fh:
    for row in csv.DictReader(fh):
        kingdoms[row["kingdom_name"]] += 1
        species_in_genus[row["genus_name"]].add(row["scientific_name"])
        if row["kingdom_name"] == "FUNGI":
            fungal_categories[row["category"]] += 1
            if row["category"] in threatened:
                threatened_fungi[row["genus_name"]] += 1

# 2. rows per kingdom
print("\nRows per kingdom:")
for kingdom, n in kingdoms.most_common():
    print(f"{kingdom}\t{n}")

# 3. genera with the most distinct species names
print("\nGenera with the most species:")
genus_sizes = {genus: len(names) for genus, names in species_in_genus.items()}
ranked = sorted(genus_sizes.items(), key=lambda pair: pair[1], reverse=True)
for genus, n in ranked[:10]:
    print(f"{genus}\t{n}")

# 4. fungi
print("\nFungi by Red List category:")
for category, n in fungal_categories.most_common():
    print(f"{category}\t{n}")
print("\nFungal genera with the most threatened (CR/EN/VU) species:")
for genus, n in threatened_fungi.most_common(5):
    print(f"{genus}\t{n}")
```

```
lines that split(',') gets wrong: 80622

Rows per kingdom:
ANIMALIA    79666
PLANTAE 63424
FUNGI   627
CHROMISTA   15

Genera with the most species:
Eucalyptus  722
Conus   628
Eugenia 530
Pristimantis    509
Syzygium    450
Diospyros   427
Miconia 421
Euphorbia   420
Myrcia  418
Quercus 412

Fungi by Red List category:
LC  211
VU  154
EN  101
DD  65
NT  61
CR  35

Fungal genera with the most threatened (CR/EN/VU) species:
Cortinarius 15
Hygrocybe   13
Amanita 9
Entoloma    8
Ramalina    5
```

Extensions: Which class of animals (`class_name`) has the most critically endangered (`CR`) species? Write the genus counts to a tab-delimited file with `csv.writer(out, delimiter="\t")`.

Problem 4: FASTA statistics

`S_cerevisiae.ORFs.fasta.gz` has the DNA sequence of every yeast open reading frame (ORF). Write a script that reports:

1. the number of sequences, and the shortest, longest and mean length,
2. the overall GC content (all G+C bases / all bases),
3. how often each **first codon** and each **last codon** occurs, most common first, as a count and a percentage,
4. how many genes are on the Watson (`W`) and Crick (`C`) strands.

Yeast systematic gene names encode their location: in `YAL001C`, `Y` = yeast, `A` = chromosome I (B = II, ...), `L` = left arm of the chromosome (`R` = right), `001` = first gene from the centromere, and `C` = Crick strand (`W` = Watson). Some names have a suffix (`YBR111W-A`), mitochondrial genes start with `Q` (`Q0010`) and genes on the 2-micron plasmid with `R` (`R0010W`). So the strand is the **7th character** of names that start with `Y`, not simply the last character; count the other names separately.

Hints

- Reuse the `read_fasta()` function from Python III: it returns a dictionary of ID -> sequence.
- `seq[:3]` is the first codon and `seq[-3:]` the last codon.

- A `Counter` for each question keeps the code short.
- Put the GC calculation in a function so you can test it: `assert gc_content("GGAT") == 0.5`.

Solution (also available as workshop_fasta_stats.py)

```python
#!/usr/bin/env python3
"""Workshop problem 4: summary statistics for the yeast ORF FASTA file."""
import gzip
from collections import Counter

fastafile = "S_cerevisiae.ORFs.fasta.gz"


def read_fasta(filename):
    """Read a FASTA file (plain or .gz) and return a dict of id -> sequence."""
    opener = gzip.open if filename.endswith(".gz") else open
    pieces = {}
    seq_id = None
    with opener(filename, "rt") as fh:
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                seq_id = line[1:].split()[0]
                pieces[seq_id] = []
            elif seq_id is not None and line:
                pieces[seq_id].append(line)
    return {sid: "".join(parts) for sid, parts in pieces.items()}


def gc_content(seq):
    """Return the fraction of G and C bases in seq (0.0 if seq is empty)."""
    if len(seq) == 0:
        return 0.0
    seq = seq.upper()
    return (seq.count("G") + seq.count("C")) / len(seq)


def strand_from_name(name):
    """Return 'W' or 'C' from a yeast systematic name, or 'other'."""
    if name.startswith("Y") and len(name) >= 7 and name[6] in "WC":
        return name[6]
    return "other"


def print_percent_table(counter, total, top=5):
    """Print the most common items of a Counter with their percentages."""
    for item, n in counter.most_common(top):
        print(f"  {item}\t{n}\t{100 * n / total:.1f}%")


if __name__ == "__main__":
    assert gc_content("GGAT") == 0.5
    assert strand_from_name("YAL001C") == "C"
    assert strand_from_name("YBR111W-A") == "W"
    assert strand_from_name("Q0010") == "other"
```

```
    seqs = read_fasta(fastafile)
    n = len(seqs)
    lengths = [len(seq) for seq in seqs.values()]
    print(f"1. {n} sequences; shortest {min(lengths)}, longest {max(lengths)},"
          f" mean {sum(lengths) / n:.1f} bp")

    all_gc = sum([seq.count("G") + seq.count("C") for seq in seqs.values()])
    print(f"2. overall GC content: {100 * all_gc / sum(lengths):.2f}%")

    first_codons = Counter([seq[:3] for seq in seqs.values()])
    last_codons = Counter([seq[-3:] for seq in seqs.values()])
    print("3. first codons:")
    print_percent_table(first_codons, n)
    print("   last codons:")
    print_percent_table(last_codons, n)

    strands = Counter([strand_from_name(name) for name in seqs])
    print(f"4. Watson (W): {strands['W']}  Crick (C): {strands['C']}"
          f"  other: {strands['other']}")
```

```
1. 6713 sequences; shortest 51, longest 14733, mean 1352.4 bp
2. overall GC content: 39.61%
3. first codons:
  ATG   6706    99.9%
  ATA   6   0.1%
  TAT   1   0.0%
   last codons:
  TAA   3160    47.1%
  TGA   1995    29.7%
  TAG   1558    23.2%
4. Watson (W): 3371  Crick (C): 3310  other: 32
```

Nearly all ORFs start with the start codon `ATG` and all end with one of the three stop codons (`TAA`, `TAG`, `TGA`), as expected. Which ORFs do not start with `ATG`? Print their IDs and look them up in SGD: six are mitochondrial (`Q...`), where `ATA` is used as a start codon (and codes for methionine), and one is a pseudogene.

Problem 5: translate the ORFs with a codon table

`codon_table.txt` has three tab-separated columns: codon, one-letter amino acid (`*` = stop) and name.

```
ATT I   Isoleucine
ATC I   Isoleucine
ATA I   Isoleucine
```

1. Write a function `read_codon_table(filename)` that returns a dictionary of codon -> amino acid.
2. Write a function `translate(dna, table)` that returns the protein sequence. Test it on `"ATGGCCTAA"` (it should give `MA*`).
3. Translate every yeast ORF. Report: how many proteins start with `M`, how many end with a stop `*`, and how many have a stop codon *inside* the protein (before the last position). Print the 5 longest proteins and their lengths (without the final `*`).

Hints

- Step through the DNA 3 bases at a time with `range(0, len(dna) - 2, 3)`; `dna[i:i + 3]` is the codon.
- Use `table.get(codon, "X")` so an unexpected codon (for example one with an `N`) gives `X` instead of a `KeyError`.
- `"*" in protein[:-1]` tests for a stop before the last position.

Solution

```python
#!/usr/bin/env python3
"""Workshop problem 5: translate yeast ORFs with a codon table."""
import gzip

fastafile = "S_cerevisiae.ORFs.fasta.gz"
codonfile = "codon_table.txt"


def read_fasta(filename):
    """Read a FASTA file (plain or .gz) and return a dict of id -> sequence."""
    opener = gzip.open if filename.endswith(".gz") else open
    pieces = {}
    seq_id = None
    with opener(filename, "rt") as fh:
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                seq_id = line[1:].split()[0]
                pieces[seq_id] = []
            elif seq_id is not None and line:
                pieces[seq_id].append(line)
    return {sid: "".join(parts) for sid, parts in pieces.items()}


def read_codon_table(filename):
    """Return a dict of codon -> one-letter amino acid."""
    table = {}
    with open(filename) as fh:
        for line in fh:
            codon, aa, name = line.rstrip("\n").split("\t")
            table[codon] = aa
    return table


def translate(dna, table):
    """Translate a DNA sequence using a codon -> amino acid dict."""
    protein = []
    for i in range(0, len(dna) - 2, 3):
        protein.append(table.get(dna[i:i + 3].upper(), "X"))
    return "".join(protein)


if __name__ == "__main__":
    table = read_codon_table(codonfile)
    assert translate("ATGGCCTAA", table) == "MA*"

    proteins = {}
    for seq_id, dna in read_fasta(fastafile).items():
        proteins[seq_id] = translate(dna, table)

    starts_m = len([p for p in proteins.values() if p.startswith("M")])
    ends_stop = len([p for p in proteins.values() if p.endswith("*")])
    internal = [sid for sid, p in proteins.items() if "*" in p[:-1]]
    print(len(proteins), "proteins")
    print(starts_m, "start with M;", ends_stop, "end with *")
```

```
    print(len(internal), "have an internal stop, e.g.", internal[:3])

    prot_len = {sid: len(p.rstrip("*")) for sid, p in proteins.items()}
    print("Longest proteins:")
    for sid in sorted(prot_len, key=prot_len.get, reverse=True)[:5]:
        print(f"  {sid}\t{prot_len[sid]} aa\t{proteins[sid][:20]}...")
```

```
6713 proteins
6706 start with M; 6711 end with *
28 have an internal stop, e.g. ['YAR061W', 'YDR134C', 'YER109C']
Longest proteins:
  YLR106C   4910 aa MSQDRILLDLDVVNQRLILF...
  YKR054C   4092 aa MCKNEARLANELIEFVAATV...
  YHR099W   3744 aa MSLTEQIEQFASRFRDDDAT...
  YDR457W   3268 aa MVLFTRCEKARKEKLAAGYK...
  YLL040C   3144 aa MLESLAANLLNRLLGSYVEN...
```

Why do 28 ORFs have internal stop codons? Print their IDs: 20 are mitochondrial genes (`Q...`). In yeast mitochondria `TGA` codes for tryptophan (W), not stop, so the standard codon table is the wrong one for them. The others are annotated in SGD as pseudogenes or "blocked reading frames" (genes broken by a stop codon in this lab strain). A good reminder to check surprising results against the biology.

Problem 6 (mini-project): ORF statistics per chromosome

Combine the GFF annotation and the ORF sequences. For each chromosome, report the number of ORFs, their mean length, and their mean GC content, and write the table to a file `orf_stats_by_chrom.tsv`.

The FASTA file has the sequences but not (in an easy-to-use way) the chromosome; the GFF file has the chromosome of every gene. The two are linked by the gene ID: `ID=YAL001C` in the GFF attributes column is `>YAL001C` in the FASTA file. This "join two files on a shared key" step is one of the most common tasks in bioinformatics, and a dictionary is the tool for it (it is what `join` does in UNIX, and a *merge* in Pandas).

Plan it as small functions:

1. `gene_chromosomes(gff_file)`: return a dict gene ID -> chromosome for all features whose type ends with `gene` (`gene`, `transposable_element_gene`, ...).
2. `read_fasta(fasta_file)`: ID -> sequence (as before).
3. Group the ORFs by chromosome: a dict of chromosome -> list of (length, GC) tuples.
4. Summarize each chromosome and write the table, in the order the chromosomes appear in the GFF file.

Hints

- The GFF attributes column looks like `ID=YAL001C;Name=YAL001C;gene=TFC3;...`. Split on `;`, then on the first `=`, to make a small dictionary of attributes: `key, value = field.split("=", 1)`.
- Count the ORF IDs that are **not** found in the GFF dictionary, so you know whether the join worked. (Here 10 are missing: six are "blocked reading frames", a separate feature type in the GFF, and four are `R...` genes from the 2-micron plasmid, which is not in the GFF file.)
- Dictionaries remember insertion order, so a dict filled while reading the GFF keeps the chromosomes in file order.

Solution

```
#!/usr/bin/env python3
"""Workshop problem 6: ORF count, mean length and GC per chromosome."""
import csv
import gzip
from collections import defaultdict

gff_file = "S_cerevisiae.gff3.gz"
fasta_file = "S_cerevisiae.ORFs.fasta.gz"
```

```
out_file = "orf_stats_by_chrom.tsv"


def parse_attributes(text):
    """Turn 'ID=YAL001C;Name=TFC3' into {'ID': 'YAL001C', 'Name': 'TFC3'}."""
    attributes = {}
    for field in text.split(";"):
        if "=" in field:
            key, value = field.split("=", 1)
            attributes[key] = value
    return attributes


def gene_chromosomes(filename):
    """Return a dict of gene ID -> chromosome from a GFF3 file."""
    gene_chrom = {}
    with gzip.open(filename, "rt") as fh:
        for line in fh:
            if line.startswith("#"):
                continue
            cols = line.rstrip("\n").split("\t")
            if len(cols) == 9 and cols[2].endswith("gene"):
                gene_id = parse_attributes(cols[8]).get("ID")
                gene_chrom[gene_id] = cols[0]
    return gene_chrom


def read_fasta(filename):
    """Read a FASTA file (plain or .gz) and return a dict of id -> sequence."""
    opener = gzip.open if filename.endswith(".gz") else open
    pieces = {}
    seq_id = None
    with opener(filename, "rt") as fh:
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                seq_id = line[1:].split()[0]
                pieces[seq_id] = []
            elif seq_id is not None and line:
                pieces[seq_id].append(line)
    return {sid: "".join(parts) for sid, parts in pieces.items()}


def gc_content(seq):
    """Return the fraction of G and C bases in seq (0.0 if seq is empty)."""
    if len(seq) == 0:
        return 0.0
    seq = seq.upper()
    return (seq.count("G") + seq.count("C")) / len(seq)


def main():
    gene_chrom = gene_chromosomes(gff_file)
    seqs = read_fasta(fasta_file)
    print(len(gene_chrom), "genes in the GFF;", len(seqs), "ORFs in the FASTA")
```

```
    # chromosome -> list of (length, gc) tuples; start with every chromosome
    # in GFF order so the output follows the order of the genome
    by_chrom = {chrom: [] for chrom in gene_chrom.values()}
    missing = []
    for seq_id, seq in seqs.items():
        if seq_id in gene_chrom:
            by_chrom[gene_chrom[seq_id]].append((len(seq), gc_content(seq)))
        else:
            missing.append(seq_id)
    print(len(missing), "ORFs not in the GFF, e.g.", missing[:4])

    with open(out_file, "w", newline="") as out:
        writer = csv.writer(out, delimiter="\t")
        writer.writerow(["chromosome", "orfs", "mean_length", "mean_gc"])
        for chrom, values in by_chrom.items():
            if len(values) == 0:
                continue
            n = len(values)
            mean_len = sum(length for length, gc in values) / n
            mean_gc = sum(gc for length, gc in values) / n
            writer.writerow([chrom, n, round(mean_len, 1), round(mean_gc, 4)])
    print("wrote", out_file)


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

```
7128 genes in the GFF; 6713 ORFs in the FASTA
10 ORFs not in the GFF, e.g. ['YDR134C', 'YER109C', 'YIL167W', 'YIR043C']
wrote orf_stats_by_chrom.tsv
```

The output file:

```
head -5 orf_stats_by_chrom.tsv
```

```
chromosome  orfs    mean_length mean_gc
chrI    120 1271.5  0.4161
chrII   462 1348.6  0.4019
chrIII  188 1217.5  0.4125
chrIV   853 1367.6  0.3967
```

Extensions

- Add a column with the number of ORFs on each strand (use the GFF strand column, or `strand_from_name()` from Problem 4).
- Take the file names from the command line with `sys.argv` (see Python II); in Python IV you will do this properly with `argparse`.
- Move `read_fasta()`, `gc_content()` and `parse_attributes()` into your own module (`seqtools.py`, see Python III) and import it in Problems 4, 5 and 6 instead of copying the functions.
- Make a plot of mean GC versus chromosome (see Plotting), or load the table into Pandas (Python IV).

Next

- The same patterns (read a table, build a dictionary keyed by ID, join and summarize) are used in the SNP workshop and the gene network workshop, and in parsing BLAST results for Homework 3.
- Python IV: Biopython reads FASTA, GenBank and many other formats for you, and Pandas turns tables like these into data frames.

- Python V: regular expressions, for pulling patterns (like gene names or motifs) out of text.