Python III: Dictionaries, Sets, Tuples and Functions

In Python I and Python II you learned to store values in variables and lists, loop over them, and read files line by line. Lists are great when order matters ("the 3rd column"). Most biological questions, though, are about *names*: "what is the sequence of gene YAL001C?", "how many genes are on chrIV?", "which genes are in both lists?". This lecture covers the data structures that answer those questions, and how to package your code into reusable **functions**.

What you'll learn

- **Dictionaries** (`dict`): look up a value by a key (gene ID -> sequence), add and update entries, test membership with `in`, use `.get()`, loop with `.items()`.
- **Counting** things with a dictionary (the Python version of `sort | uniq -c`), then the shortcuts `Counter` and `defaultdict`.
- **Sorting** a dictionary by its values with `sorted(..., key=...)`.
- **Tuples** (fixed groups of values) and **unpacking**.
- **Sets** for unique items and for comparing lists (intersection, union, difference).
- **Nested structures**: a dictionary of lists, a list of dictionaries.
- **Functions**: `def`, parameters, `return`, default and keyword arguments, docstrings, local variables, testing, and `if __name__ == "__main__":`.
- Writing your own **module** and `import`ing it.
- A clean **FASTA reader** that returns a dictionary of ID -> sequence.

Getting the data

The examples use the yeast (*Saccharomyces cerevisiae*) genome annotation (GFF3) and the yeast open reading frames (ORFs, the protein-coding DNA sequences) in FASTA format. Download them in the shell into the folder where you will run Python:

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

The `.gz` files stay compressed; we open them with `gzip.open()` as shown in Python II.

Dictionaries: looking things up by name

A **dictionary** stores **key: value** pairs. You look up a value by its key instead of by its position. Keys are usually strings (gene IDs, chromosome names, codons) or numbers; values can be anything. Create one with curly braces `{}`:

```
gene_length = {"YAL001C": 3483, "YAL002W": 3825, "YAL003W": 621}
print(gene_length)
print(gene_length["YAL002W"])   # look up a value by its key
print(len(gene_length))         # number of key:value pairs
```

```
{'YAL001C': 3483, 'YAL002W': 3825, 'YAL003W': 621}
3825
3
```

Add a new entry, or change an existing one, by assigning to a key. Each key can appear only once, so assigning to an existing key **replaces** the old value:

```
gene_length = {"YAL001C": 3483, "YAL002W": 3825}
gene_length["YAL003W"] = 621     # add a new key
gene_length["YAL001C"] = 3480    # replace the value for an existing key
print(gene_length)
```

```
del gene_length["YAL002W"]       # remove a key (and its value)
print(gene_length)
```

```
{'YAL001C': 3480, 'YAL002W': 3825, 'YAL003W': 621}
{'YAL001C': 3480, 'YAL003W': 621}
```

Asking for a key that is not in the dictionary is an error (a `KeyError`):

```
gene_length = {"YAL001C": 3483, "YAL002W": 3825}
print(gene_length["YFG1"])
```

```
Traceback (most recent call last):
  ...
KeyError: 'YFG1'
```

There are two ways to avoid this. Test first with `in`, or use `.get(key, default)`, which returns the default (or `None`) when the key is missing:

```
gene_length = {"YAL001C": 3483, "YAL002W": 3825}
for gene in ["YAL001C", "YFG1"]:
    if gene in gene_length:
        print(gene, "length is", gene_length[gene])
    else:
        print(gene, "is not in the dictionary")

print(gene_length.get("YAL001C"))
print(gene_length.get("YFG1"))       # missing key -> None
print(gene_length.get("YFG1", 0))    # missing key -> the default we gave
```

```
YAL001C length is 3483
YFG1 is not in the dictionary
3483
None
0
```

A dictionary is a lookup table. A codon table is a perfect example: the key is a codon and the value is the amino acid.

```
codon_table = {"ATG": "M", "TGG": "W", "TTT": "F", "TAA": "*"}
dna = "ATGTTTTGGTAA"
for i in range(0, len(dna), 3):     # step through the sequence 3 at a time
    codon = dna[i:i + 3]
    print(codon, "->", codon_table[codon])
```

```
ATG -> M
TTT -> F
TGG -> W
TAA -> *
```

Notes:

- Dictionaries remember the order in which keys were added (Python 3.7 and later), so printing or looping shows keys in insertion order.
- Keys must be *immutable* (unchangeable) values: strings, numbers or tuples. A list cannot be a key.
- Checking `key in d` is fast, even for millions of keys. Checking `item in some_list` has to scan the whole list, so it gets slow for large lists.

Looping over a dictionary

Looping directly over a dictionary gives you the **keys**. `.values()` gives the values and `.items()` gives (key, value) pairs, which you can unpack into two loop variables:

```python
chrom_size = {"chrI": 230218, "chrII": 813184, "chrIII": 316620}

for chrom in chrom_size:              # same as: for chrom in chrom_size.keys():
    print(chrom)

print(sum(chrom_size.values()), "bp in total")

for chrom, size in chrom_size.items():
    print(f"{chrom}\t{size}")
```

```
chrI
chrII
chrIII
1360022 bp in total
chrI    230218
chrII   813184
chrIII  316620
```

You can also build a dictionary from two parallel lists with `zip()` (see Python II), or with a *dictionary comprehension*, which works like a list comprehension but with `key: value`:

```python
genes = ["YAL001C", "YAL002W", "YAL003W"]
lengths = [3483, 3825, 621]
gene_length = dict(zip(genes, lengths))
print(gene_length)

long_genes = {g: n for g, n in gene_length.items() if n > 1000}
print(long_genes)
```

```
{'YAL001C': 3483, 'YAL002W': 3825, 'YAL003W': 621}
{'YAL001C': 3483, 'YAL002W': 3825}
```

Do not add or remove keys from a dictionary *while* you loop over it; Python stops with `RuntimeError: dictionary changed size during iteration`. Build a new dictionary instead (as in the comprehension above).

Counting with a dictionary

Counting is the most common thing you will do with a dictionary: how many times does each base, codon, chromosome, or species appear? The key is the thing you are counting and the value is its count. In the UNIX lectures you did this with `sort | uniq -c` or with an awk array, `awk '{count[$1]++} END {for (k in count) print k, count[k]}'` (see the UNIX data processing lab). A Python dictionary is the same idea.

Version 1: test with `in`. The first time we see a key we set its count to 1; after that we add 1.

```python
dna = "ATGGCGTTAGCATTAGGC"
base_count = {}
for base in dna:
    if base in base_count:
        base_count[base] += 1
    else:
        base_count[base] = 1
print(base_count)
```

```
{'A': 4, 'T': 5, 'G': 6, 'C': 3}
```

Version 2: `.get(key, 0)`. `.get()` returns 0 for a key we have not seen yet, so one line does the same job:

```python
dna = "ATGGCGTTAGCATTAGGC"
base_count = {}
```

```
for base in dna:
    base_count[base] = base_count.get(base, 0) + 1
print(base_count)
```

```
{'A': 4, 'T': 5, 'G': 6, 'C': 3}
```

Version 3: `collections.Counter`. Counting is so common that the standard library has a dictionary made for it. `Counter` counts any list, string or other iterable, returns 0 for missing keys, and has a `.most_common(n)` method:

```
from collections import Counter

dna = "ATGGCGTTAGCATTAGGC"
base_count = Counter(dna)
print(base_count)
print(base_count["G"], base_count["N"])   # N is not there -> 0, no error
print(base_count.most_common(2))
```

```
Counter({'G': 6, 'T': 5, 'A': 4, 'C': 3})
6 0
[('G', 6), ('T', 5)]
```

All three give the same answer; use whichever you find clearest. It is worth understanding version 1, because the same "have I seen this key before?" logic appears everywhere.

Real data: genes per chromosome from a GFF file

A GFF3 file has 9 tab-separated columns: chromosome, source, feature type, start, end, score, strand, phase, attributes. Lines starting with `#` are comments. Let's count the `gene` features on each chromosome:

```
import gzip

genes_per_chrom = {}
with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        if line.startswith("#"):
            continue
        cols = line.rstrip("\n").split("\t")
        if len(cols) != 9 or cols[2] != "gene":
            continue
        chrom = cols[0]
        genes_per_chrom[chrom] = genes_per_chrom.get(chrom, 0) + 1

for chrom, n in genes_per_chrom.items():
    print(chrom, n)
print("total genes:", sum(genes_per_chrom.values()))
```

```
chrI 117
chrII 456
chrIII 184
chrIV 836
chrV 323
chrVI 139
chrVII 583
chrVIII 321
chrIX 241
chrX 398
chrXI 348
chrXII 578
```

```
chrXIII 505
chrXIV 435
chrXV 597
chrXVI 511
chrmt 28
total genes: 6600
```

Compare with the shell version: `zcat S_cerevisiae.gff3.gz | awk -F'\t' '$3=="gene"{n[$1]++} END{for(c in n) print c, n[c]}'`. The Python version is longer, but it is easy to extend (count genes per chromosome *and* strand, skip dubious ORFs, compute lengths…).

Sorting a dictionary

`sorted()` on a dictionary returns a list of its **keys** in sorted order (alphabetical for strings, smallest to largest for numbers):

```
counts = {"TAA": 3, "TGA": 5, "TAG": 1}
print(sorted(counts))
for codon in sorted(counts):
    print(codon, counts[codon])
```

```
['TAA', 'TAG', 'TGA']
TAA 3
TAG 1
TGA 5
```

Often we want the keys ordered by their **values** instead (most common codon first). `sorted()` has an optional `key=` argument: a *function* that is applied to each item, and whose result is used for sorting. You have already seen a simple one: `sorted(numbers_as_text, key=int)` sorts strings like `"141"` and `"7"` as numbers:

```
numbers = ["141", "7", "90", "3"]
print(sorted(numbers))            # sorted as text: "1" comes before "3"
print(sorted(numbers, key=int))   # sorted as numbers
print(sorted(numbers, key=int, reverse=True))
```

```
['141', '3', '7', '90']
['3', '7', '90', '141']
['141', '90', '7', '3']
```

`counts.items()` gives `(key, value)` pairs like `("TGA", 5)`. To sort the pairs by the value we need a function that takes one pair and returns its second element, `pair[1]`. A **lambda** is a small, one-line, unnamed function written right where it is needed: `lambda pair: pair[1]` means "given `pair`, return `pair[1]`".

```
counts = {"TAA": 3, "TGA": 5, "TAG": 1}
by_count = sorted(counts.items(), key=lambda pair: pair[1], reverse=True)
print(by_count)
for codon, n in by_count:
    print(codon, n)
```

```
[('TGA', 5), ('TAA', 3), ('TAG', 1)]
TGA 5
TAA 3
TAG 1
```

The lambda above is just a short way of writing this regular function (functions are explained below):

```
def second_item(pair):
    return pair[1]

counts = {"TAA": 3, "TGA": 5, "TAG": 1}
print(sorted(counts.items(), key=second_item, reverse=True))
```

```
[('TGA', 5), ('TAA', 3), ('TAG', 1)]
```

Another common idiom sorts the keys by looking up their values: `sorted(counts, key=counts.get, reverse=True)`. And a `Counter` already knows how: `Counter(...).most_common()`.

Chromosomes with the most genes, using the GFF counting code from above:

```
import gzip

genes_per_chrom = {}
with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        cols = line.rstrip("\n").split("\t")
        if len(cols) == 9 and cols[2] == "gene":
            genes_per_chrom[cols[0]] = genes_per_chrom.get(cols[0], 0) + 1

ranked = sorted(genes_per_chrom.items(), key=lambda pair: pair[1], reverse=True)
for chrom, n in ranked[:3]:          # the top 3
    print(chrom, n)
```

```
chrIV 836
chrXV 597
chrVII 583
```

Keeping the best value for each key

Another very common pattern: keep only the *largest* (or smallest) value seen for each key, for example the best BLAST hit for each query sequence, or the longest transcript of each gene. Store the best so far and replace it when something better comes along:

```
# query, subject, percent identity (like columns 1-3 of BLAST tabular output)
hits = [("geneA", "ref1", 99.1), ("geneA", "ref7", 85.0),
        ("geneB", "ref2", 72.4), ("geneB", "ref9", 78.9),
        ("geneC", "ref3", 100.0)]

best = {}                            # query -> (subject, pident) of its best hit
for query, subject, pident in hits:
    if query not in best or pident > best[query][1]:
        best[query] = (subject, pident)

for query in sorted(best):
    subject, pident = best[query]
    print(query, subject, pident)
```

```
geneA ref1 99.1
geneB ref9 78.9
geneC ref3 100.0
```

The same `key=` idea sorts rows of a table by any column, for example to list the lowest-identity best hits first:

```
best = {"geneA": ("ref1", 99.1), "geneB": ("ref9", 78.9),
        "geneC": ("ref3", 100.0)}
rows = [(query, subject, pident) for query, (subject, pident) in best.items()]
for row in sorted(rows, key=lambda row: row[2]):
    print(row)
```

```
('geneB', 'ref9', 78.9)
('geneA', 'ref1', 99.1)
('geneC', 'ref3', 100.0)
```

Tuples and unpacking

A **tuple** is like a list, but written with parentheses and it cannot be changed after it is created (it is *immutable*). Use a tuple for a small, fixed group of values that belong together, such as a genomic interval:

```
exon = ("chrI", 1807, 2169, "-")      # chromosome, start, end, strand
print(exon[0], exon[2] - exon[1] + 1)
chrom, start, end, strand = exon      # "unpacking" into 4 variables
print(chrom, strand)
```

```
chrI 363
chrI -
```

Trying to change a tuple fails:

```
exon = ("chrI", 1807, 2169, "-")
exon[1] = 1800
```

```
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

You have already been using tuples: `.items()` gives `(key, value)` tuples, and `for chrom, n in d.items():` unpacks each one. Unpacking also gives a neat way to swap two values: `a, b = b, a`.

Because tuples are immutable, they can be dictionary keys. That lets you count combinations, for example genes per (chromosome, strand):

```
genes = [("chrI", "+"), ("chrI", "-"), ("chrI", "+"), ("chrII", "-")]
counts = {}
for chrom, strand in genes:
    key = (chrom, strand)
    counts[key] = counts.get(key, 0) + 1
print(counts)
print(counts[("chrI", "+")])
```

```
{('chrI', '+'): 2, ('chrI', '-'): 1, ('chrII', '-'): 1}
2
```

Sets: unique items and comparing lists

A **set** is an unordered collection with **no duplicates**. Make one with `set(some_list)` or with curly braces; `set()` is an empty set (`{}` is an empty *dictionary*). Sets are the Python version of `sort -u`, and of the UNIX `comm` command for comparing two lists.

```
chroms = ["chrI", "chrII", "chrI", "chrIII", "chrII"]
unique = set(chroms)
print(len(unique), sorted(unique))  # sets have no order; sort to print
unique.add("chrIV")
print("chrIV" in unique)
```

```
3 ['chrI', 'chrII', 'chrIII']
True
```

Set operations compare two collections. Suppose we have genes that are up-regulated in two experiments:

```
heat = {"HSP104", "HSP82", "SSA1", "HSP26", "CTT1"}
salt = {"CTT1", "GPD1", "HSP26", "ENA1"}

print(sorted(heat & salt))   # intersection: in both
print(sorted(heat | salt))   # union: in either
```

```
print(sorted(heat - salt))   # difference: in heat but not in salt
print(sorted(heat ^ salt))   # in one or the other, but not both
```

```
['CTT1', 'HSP26']
['CTT1', 'ENA1', 'GPD1', 'HSP104', 'HSP26', 'HSP82', 'SSA1']
['HSP104', 'HSP82', 'SSA1']
['ENA1', 'GPD1', 'HSP104', 'HSP82', 'SSA1']
```

The same operations are available as methods, which also accept lists: `heat.intersection(salt_list)`, `heat.union(...)`, `heat.difference(...)`.

Nested data: dictionaries of lists, lists of dictionaries

The value in a dictionary can be any object, including a list or another dictionary. This is how you represent one-to-many relationships, such as a gene with several exons.

A dictionary of lists (gene -> list of exon lengths). The first time we see a gene we create an empty list, then we append to it:

```
exons = [("YFL039C", 1118), ("YAL003W", 80), ("YFL039C", 10),
         ("YAL003W", 541), ("YBR111W-A", 71), ("YBR111W-A", 140),
         ("YBR111W-A", 80), ("YAL001C", 3413), ("YAL001C", 70)]

exons_by_gene = {}
for gene, length in exons:
    if gene not in exons_by_gene:
        exons_by_gene[gene] = []
    exons_by_gene[gene].append(length)

for gene, lengths in exons_by_gene.items():
    print(gene, len(lengths), "exons", sum(lengths), "bp", lengths)
```

```
YFL039C 2 exons 1128 bp [1118, 10]
YAL003W 2 exons 621 bp [80, 541]
YBR111W-A 3 exons 291 bp [71, 140, 80]
YAL001C 2 exons 3483 bp [3413, 70]
```

`collections.defaultdict(list)` creates the empty list automatically the first time a key is used, so the `if` is not needed. (`defaultdict(int)` does the same for counting, starting at 0.)

```
from collections import defaultdict

exons = [("YFL039C", 1118), ("YAL003W", 80), ("YFL039C", 10)]
exons_by_gene = defaultdict(list)
for gene, length in exons:
    exons_by_gene[gene].append(length)
print(dict(exons_by_gene))
```

```
{'YFL039C': [1118, 10], 'YAL003W': [80]}
```

A list of dictionaries is a natural way to hold a table where each row is a record with named fields. This is what `csv.DictReader` gives you (one dictionary per row, keyed by the column names in the header), and what Pandas turns into a table in Python IV:

```
genes = [
    {"id": "YAL001C", "chrom": "chrI", "length": 3483},
    {"id": "YBR111W-A", "chrom": "chrII", "length": 291},
    {"id": "YAL003W", "chrom": "chrI", "length": 621},
]
for gene in genes:
```

```
    if gene["chrom"] == "chrI":
        print(gene["id"], gene["length"])
longest = max(genes, key=lambda g: g["length"])
print("longest:", longest["id"])
```

```
YAL001C 3483
YAL003W 621
longest: YAL001C
```

Real data: exons per gene from the GFF file

In the yeast GFF, each `CDS` line has an attribute like `Parent=YFL039C_mRNA`. Collecting CDS lengths per mRNA finds the genes that have introns (more than one CDS piece). We set `parent = None` for each line so a value from the previous line can never be reused by mistake:

```
import gzip
from collections import defaultdict

cds_lengths = defaultdict(list)
with gzip.open("S_cerevisiae.gff3.gz", "rt") as fh:
    for line in fh:
        cols = line.rstrip("\n").split("\t")
        if len(cols) != 9 or cols[2] != "CDS":
            continue
        start, end = int(cols[3]), int(cols[4])
        parent = None
        for field in cols[8].split(";"):          # e.g. Parent=YFL039C_mRNA
            if field.startswith("Parent="):
                parent = field[len("Parent="):]
        cds_lengths[parent].append(end - start + 1)

multi = {mrna: lens for mrna, lens in cds_lengths.items() if len(lens) > 1}
print(len(cds_lengths), "transcripts;", len(multi), "have >1 CDS piece")
print("YFL039C_mRNA", cds_lengths["YFL039C_mRNA"])
```

```
6697 transcripts; 331 have >1 CDS piece
YFL039C_mRNA [1118, 10]
```

Functions: naming a piece of work

A **function** is a named block of code that takes some input (**parameters**), does a job, and gives back a result with **return**. You have been calling built-in functions all along: `len()`, `print()`, `sorted()`, `int()`. Writing your own lets you:

- reuse the same code many times without copy-and-paste (and fix a bug in one place),
- give a job a name, which makes the main program read like a recipe,
- test small pieces separately.

Define a function with `def`, a name, parameters in parentheses, and a colon. The body is indented, just like the body of a loop:

```
def gc_content(seq):
    """Return the fraction of G and C bases in a DNA sequence."""
    seq = seq.upper()
    gc = seq.count("G") + seq.count("C")
    return gc / len(seq)

print(gc_content("ATGC"))
print(gc_content("ggccat"))
```

```
value = gc_content("ATGCGCGCAA")
print(f"GC = {value:.1%}")
```

```
0.5
0.6666666666666666
GC = 60.0%
```

Things to notice:

- `seq` is a **parameter**: a variable that receives whatever value is passed in when the function is *called* (`"ATGC"`, then `"ggccat"`).
- The string in triple quotes right after the `def` line is a **docstring**. It documents what the function does; `help(gc_content)` prints it.
- `return` sends a value back to the caller and ends the function. The code that calls the function decides what to do with it: print it, store it, or use it in a calculation.
- Defining a function does not run it. The body runs only when the function is called.

return versus print

A common beginner mistake is to `print` inside a function instead of returning. Printing shows a value on the screen but gives nothing back to the program; the function then returns `None`:

```
def gc_print(seq):
    print((seq.count("G") + seq.count("C")) / len(seq))

def gc_return(seq):
    return (seq.count("G") + seq.count("C")) / len(seq)

a = gc_print("ATGC")      # prints 0.5, but a gets nothing
b = gc_return("ATGC")     # prints nothing, b gets 0.5
print("a =", a, " b =", b)
```

```
0.5
a = None  b = 0.5
```

Let functions **return** results and do the printing in the main part of the program. Then the same function can be used to print a report, write a file, or feed another calculation.

Several parameters, default values and keyword arguments

A function can take several parameters. Give a parameter a **default value** with `=` to make it optional:

```
def count_kmers(seq, k=2):
    """Count every overlapping k-mer (default k=2) in seq."""
    counts = {}
    for i in range(len(seq) - k + 1):
        kmer = seq[i:i + k]
        counts[kmer] = counts.get(kmer, 0) + 1
    return counts

print(count_kmers("ATATGC"))             # uses the default k=2
print(count_kmers("ATATGC", 3))          # positional: k=3
print(count_kmers(seq="ATATGC", k=3))    # keyword arguments, by name
```

```
{'AT': 2, 'TA': 1, 'TG': 1, 'GC': 1}
{'ATA': 1, 'TAT': 1, 'ATG': 1, 'TGC': 1}
{'ATA': 1, 'TAT': 1, 'ATG': 1, 'TGC': 1}
```

Keyword arguments (`name=value`) make calls easier to read and let you skip over optional parameters. You have already used them: `sorted(x, reverse=True)`, `print(a, b, sep="\t")`.

Returning several values

A function returns one object, but that object can be a tuple, which the caller can unpack:

```
def length_stats(lengths):
    """Return (count, minimum, maximum, mean) for a list of numbers."""
    n = len(lengths)
    return n, min(lengths), max(lengths), sum(lengths) / n

n, shortest, longest, mean = length_stats([621, 3483, 3825, 291])
print(n, shortest, longest, mean)
```

```
4 291 3825 2055.0
```

Scope: variables inside a function are local

Variables created inside a function (including its parameters) are **local**: they exist only while the function runs and are separate from variables with the same name elsewhere.

```
total = 100

def add_lengths(lengths):
    total = 0                   # a new, local variable called total
    for n in lengths:
        total += n
    return total

print(add_lengths([1, 2, 3]))
print(total)                    # the outer total is unchanged
```

```
6
100
```

This is a good thing: a function works only with what you pass in and hands back what it returns, so you can understand it on its own. Pass data in as parameters rather than relying on variables defined outside the function.

Small functions are easy to test

Because a function has clear inputs and outputs, you can check it with a few cases where you know the answer. `assert` stops the program with an `AssertionError` if its condition is `False`:

```
def reverse_complement(seq):
    """Return the reverse complement of a DNA sequence."""
    complement = {"A": "T", "T": "A", "G": "C", "C": "G", "N": "N"}
    rc = ""
    for base in reversed(seq.upper()):
        rc += complement[base]
    return rc

assert reverse_complement("ATGC") == "GCAT"
assert reverse_complement("aaa") == "TTT"
assert reverse_complement("") == ""
print("all tests passed")
print(reverse_complement("ATGGCGTTAG"))
```

```
all tests passed
CTAACGCCAT
```

Testing small functions like this is how the GitHub Classroom autograder checks your homework: it imports your function and calls it with known inputs.

Scripts, modules and `if __name__ == "__main__":`

Any `.py` file is also a **module** that other scripts can `import`. Put your useful functions in a file, for example `seqtools.py`:

```
"""seqtools: small helper functions for DNA sequences."""


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


def reverse_complement(seq):
    """Return the reverse complement of a DNA sequence."""
    complement = {"A": "T", "T": "A", "G": "C", "C": "G", "N": "N"}
    rc = ""
    for base in reversed(seq.upper()):
        rc += complement[base]
    return rc


if __name__ == "__main__":
    # runs only when this file is run as a script: python3 seqtools.py
    print(gc_content("ATGC"), reverse_complement("ATGC"))
```

Then, in another script **in the same folder**, import it and call its functions as `module.function()`, or import just the names you need:

```
import seqtools
print(seqtools.gc_content("GGGCCCAT"))

from seqtools import reverse_complement
print(reverse_complement("ATGAAA"))
```

```
0.75
TTTCAT
```

What does `if __name__ == "__main__":` do? When Python runs a file directly (`python3 seqtools.py`), it sets the special variable `__name__` to `"__main__"`. When the file is imported, `__name__` is the module name (`"seqtools"`) instead. So the code under that `if` runs when you use the file as a program, but not when another script imports its functions:

```
python3 seqtools.py
```

```
0.5 GCAT
```

A good layout for your homework scripts is: imports at the top, then functions, then the main program under `if __name__ == "__main__":` (often in a function called `main()`). The file avg.py in this folder is a small example you can both run and import (`from avg import average`):

```
python3 avg.py
python3 -c 'from avg import average; print(average([10, 20, 60]))'
```

```
mean exon length: 159.8 bp
30.0
```

Reading a FASTA file into a dictionary

FASTA files have a header line starting with `>` followed by one or more sequence lines:

```
>YAL001C TFC3 SGDID:S000000001, Chr I from 151006-147594, ...
ATGGTACTGACGATTTATCCTGACGAACTCGTACAAATAGTGTCTGATAAAATTGCTTCA
AATAAGGGAAAAATCACTTTGAATCAGCTGTGGGATATATCTGGTAAATATTTTGATTTG
```

To read it, loop over the lines and keep track of the *current* sequence ID. When a line starts with `>`, take the first word after the `>` as the ID and start an empty sequence; otherwise add the line to the current sequence. We store the pieces in a list and join them at the end, which is faster than adding strings over and over:

```
import gzip


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


seqs = read_fasta("S_cerevisiae.ORFs.fasta.gz")
print(len(seqs), "sequences")
print(seqs["YAL003W"][:60])
print(len(seqs["YAL003W"]), "bp")
```

```
6713 sequences
ATGGCATCCACCGATTTCTCCAAGATTGAAACTTTGAAACAATTAAACGCTTCTTTGGCT
621 bp
```

Now the dictionary makes the analysis short. GC content and length of the first few sequences, and summary statistics for all of them:

```
for seq_id in list(seqs)[:3]:          # the first 3 IDs
    seq = seqs[seq_id]
    gc = (seq.count("G") + seq.count("C")) / len(seq)
    print(f"{seq_id}\t{len(seq)}\t{gc:.3f}")

lengths = [len(s) for s in seqs.values()]
print("shortest", min(lengths), "longest", max(lengths),
      "mean", round(sum(lengths) / len(lengths), 1))
```

```
YAL001C 3483    0.371
YAL002W 3825    0.372
YAL003W 621 0.446
shortest 51 longest 14733 mean 1352.4
```

The last line is a dictionary comprehension that joins each list of lines into one string. The versions below write the file opening in one line, `opener = gzip.open if filename.endswith(".gz") else open`: a short if/else that picks *which function* to call, so that `opener(filename, "rt")` can be used in a `with` statement.

A generator version with `yield` (optional)

`read_fasta()` keeps every sequence in memory, which is fine for a yeast genome but not for a 50 GB sequencing file. A **generator** function uses `yield` instead of `return`: each `yield` hands back one value (here an `(id, sequence)` tuple) and pauses the function until the loop asks for the next one. Only one record is in memory at a time.

```
import gzip


def fasta_records(filename):
    """Yield (id, sequence) tuples from a FASTA file, one at a time."""
    opener = gzip.open if filename.endswith(".gz") else open
    with opener(filename, "rt") as fh:
        seq_id, parts = None, []
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                if seq_id is not None:
                    yield seq_id, "".join(parts)   # finish the previous record
                seq_id, parts = line[1:].split()[0], []
            elif line:
                parts.append(line)
        if seq_id is not None:
            yield seq_id, "".join(parts)           # the last record in the file


n = 0
total = 0
for seq_id, seq in fasta_records("S_cerevisiae.ORFs.fasta.gz"):
    n += 1
    total += len(seq)
print(n, "sequences,", total, "bp")
# or collect all the records into a dictionary:
seqs = dict(fasta_records("S_cerevisiae.ORFs.fasta.gz"))
print(len(seqs))
```

```
6713 sequences, 9078756 bp
6713
```

Writing your own parser is a good exercise, and you will see older code (for example in our code templates) that does the same with `itertools.groupby`. In practice, for FASTA, FASTQ, GenBank and other formats you will usually use Biopython's `SeqIO.parse()`, covered in Python IV.

Putting it together: codon usage and translation

`codon_table.txt` has three tab-separated columns: codon, one-letter amino acid (`*` for stop) and amino acid name. Read it into a dictionary with a function, then use it to translate an ORF and to count codon usage across the whole yeast genome.

```python
from collections import Counter
import gzip


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


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()}


table = read_codon_table("codon_table.txt")
print(len(table), "codons; ATG ->", table["ATG"], "; TAA ->", table["TAA"])

seqs = read_fasta("S_cerevisiae.ORFs.fasta.gz")
print(translate(seqs["YAL003W"], table)[:50])

codon_usage = Counter()
for seq in seqs.values():
    for i in range(0, len(seq) - 2, 3):
        codon_usage[seq[i:i + 3]] += 1

total = sum(codon_usage.values())
print(total, "codons counted")
for codon, n in codon_usage.most_common(5):
    print(f"{codon}\t{table.get(codon, '?')}\t{n}\t{n / total:.2%}")
```

```
64 codons; ATG -> M ; TAA -> *
MASTDFSKIETLKQLNASLADKSYIEGTAVSQADVTVFKAFQSAYPEFSR
```

```
3026251 codons counted
GAA E   136081  4.50%
AAA K   128872  4.26%
GAT D   113203  3.74%
AAT N   110253  3.64%
ATT I   91270   3.02%
```

Common mistakes

- `KeyError`: looking up a key that is not there. Check with `if key in d:` or use `d.get(key, default)`. Also check for invisible differences: `"YAL001C\n"` is not the same key as `"YAL001C"` (use `.strip()`), and `"chrI"` is not `"ChrI"`.
- **Numbers read from files are strings.** `"100" + "20"` is `"10020"`, and `sorted(["141", "7"])` sorts as text. Convert with `int()` or `float()` before doing math or sorting by value.
- **`{}` is an empty dictionary, not an empty set.** Use `set()`.
- **Forgetting `return`.** A function without `return` gives back `None`, so `x = f(...)` sets `x` to `None`.
- **Calling a function before it is defined** in the file gives `NameError`. Put `def` blocks above the code that uses them.
- **Using a list as a key** gives `TypeError: unhashable type: 'list'`; use a tuple.
- **Resetting inside the loop**: `counts = {}` inside the `for` loop starts over on every line. Create the dictionary before the loop.
- **Naming a variable like a built-in** (`list`, `dict`, `sum`, `str`, `max`) hides the built-in and causes confusing errors later, for example `TypeError: 'list' object is not callable`.
- **Naming your file like a module** you import (`csv.py`, `gzip.py`, `random.py`): `import csv` then imports your file instead.

Quick reference

Task	Code
empty dict / set	`d = {}` / `s = set()`
create dict	`d = {"chrI": 230218, "chrII": 813184}`
look up / add or replace	`d[key]` / `d[key] = value`
lookup with default	`d.get(key, 0)`
test membership	`if key in d:`
remove a key	`del d[key]`
number of keys	`len(d)`
loop over keys / values / pairs	`for k in d:` / `d.values()` / `for k, v in d.items():`
count	`d[k] = d.get(k, 0) + 1` or `Counter(items)`
group into lists	`groups = defaultdict(list)`; `groups[k].append(x)`
keys sorted	`sorted(d)`
pairs sorted by value, largest first	`sorted(d.items(), key=lambda kv: kv[1], reverse=True)`
top n of a Counter	`c.most_common(n)`
tuple and unpacking	`t = ("chrI", 100, 200)`; `chrom, start, end = t`
unique items	`set(items)`
in both / either / only first	`a & b` / `a.union(b)` / `a - b`
define a function	`def name(param, opt=default): ... return value`
call with keywords	`name(value, opt=3)`
import your module	`import seqtools` / `from seqtools import gc_content`
main program guard	`if __name__ == "__main__":`

Exercises

Use the yeast files from Getting the data.

1. Make a dictionary of the one-letter code for 5 amino acids to their full names (`"M": "Methionine"`, ...). Loop over `"MKWG"` and print the full name of each letter, printing `unknown` for a letter that is not in your dictionary (use `.get()`).
2. Count the bases in `"ATGNNCGTAGCAATTTGA"` three ways: with `if key in d`, with `.get()`, and with `Counter`. Print the counts in alphabetical order of the base.
3. Write a function `gc_content(seq)` that returns the fraction of GC. Add `assert` tests for `"GGCC"` (1.0), `"ATAT"` (0.0) and an empty string (decide what it should return). Then use it to print the GC content of every sequence in the yeast ORF file that is longer than 10,000 bp.
4. From the GFF file, count how many features of each type (column 3) there are and print them from most to least common.
5. Genes per chromosome **and strand**: count `gene` features using a `(chromosome, strand)` tuple as the key. Which chromosome has the largest difference between the `+` and `-` strands?
6. Two sets: make a set of the IDs of `gene` features in the GFF (the `ID=` attribute) and a set of the IDs in the ORF FASTA file. How many are in both? How many only in the GFF? Print a few examples of each.
7. Write a module `seqtools.py` with `read_fasta()`, `gc_content()` and `translate()` and an `if __name__ == "__main__":` block that runs a few `assert` tests. In a second script, import it, translate all yeast ORFs, and report how many proteins do **not** start with `M`, and how many have a stop (`*`) *before* the last codon.
8. Length histogram: using `read_fasta()`, put each ORF length into a 1,000 bp bin (`length // 1000 * 1000`) and count sequences per bin with a dictionary. Print every bin from 0 to the largest bin, in order, including bins with a count of 0 (hint: `range()` and `.get(bin, 0)`). Write a function that returns the dictionary and takes the bin size as a parameter with a default of 1000.

Next

- Practice with the Python workshop, which combines Python I-III on real data.
- Python IV: the standard library, `argparse`, installing packages, Biopython (`SeqIO`) and Pandas.
- Dictionaries are used heavily in the SNP workshop and the gene network workshop.
- The Python tutorial on data structures: https://docs.python.org/3/tutorial/datastructures.html and on functions: https://docs.python.org/3/tutorial/controlflow.html#defining-functions