Regular Expressions: Worked Examples

These worked examples go with Python V: Regular Expressions. Each one starts from a real data-processing problem, builds the pattern step by step, shows real output, and points out the traps. The solutions to the Python V exercises are at the end.

What's here

Setup

Download the data into a working folder (on the cluster, e.g. `~/bigdata/gen220/python5`) and run the examples in order in one Jupyter notebook or `python3` session:

```
URL=https://github.com/biodataprog/GEN220_data/raw/main/genome
curl -L -O $URL/S_cerevisiae.ORFs.fasta.gz
curl -L -O $URL/S_cerevisiae.gff3.gz
curl -L -O $URL/S_cerevisiae.fasta.gz
curl -L -O $URL/E_coli_K12.pep.gz
```

All the examples use these imports:

```
import re
import gzip
from collections import Counter
```

`Counter` (from the `collections` module) is a dictionary that counts things: `counts[key] += 1` works even the first time a key is seen, and `counts.most_common()` lists the keys from most to least common.

Example 1: SGD yeast ORF headers

Every sequence in `S_cerevisiae.ORFs.fasta.gz` has a header like this (the description is cut short here):

```
>YAL001C TFC3 SGDID:S000000001, Chr I from 151006-147594,151166-151097, Genome
Release 64-2-1, reverse complement, intron sequence removed, Verified ORF,
"Subunit of RNA polymerase III transcription initiation factor complex; ..."
```

We want a table with one row per ORF: systematic name, gene name, SGD ID, chromosome, strand, exon coordinates, ORF type (Verified, Dubious ...) and the description.

Step 1: look at the structure

It is always worth looking at several headers before you write a pattern:

```
with gzip.open("S_cerevisiae.ORFs.fasta.gz", "rt") as fh:
    headers = [line.rstrip("\n") for line in fh if line.startswith(">")]
print(len(headers), "headers")
for h in [headers[1], headers[6712]]:
    print(h[:85])
```

```
6713 headers
>YAL002W VPS8 SGDID:S000000002, Chr I from 143707-147531, Genome Release 64-2-1, Veri
>R0040C REP2 SGDID:S000029676, 2-micron plasmid from 6198-5308, Genome Release 64-2-1
```

The header has three parts:

- the start: `>NAME GENE SGDID:ID,` - space separated, easy
- a middle part of **comma separated fields**: the location, the genome release, optional notes such as `reverse complement` or `intron sequence removed`, and the ORF type
- the description in double quotes, which can itself contain commas, semicolons and even more quotes

A single giant regex for all of this would be hard to read. Instead, use a regex to cut the header into the three parts, and `split` for the simple middle part.

Step 2: one pattern for the overall shape

```
header_pat = re.compile(r'^>(\S+) (\S+) SGDID:(S\d+), (.*?), "(.*)"$')

m = header_pat.match(headers[0])
orf, gene, sgdid, fields, desc = m.groups()
print(orf, gene, sgdid)
print(fields[:50] + " ...")
print(desc[:50] + " ...")
```

```
YAL001C TFC3 S000000001
Chr I from 151006-147594,151166-151097, Genome Rel ...
Subunit of RNA polymerase III transcription initia ...
```

- `(\S+)` is "one or more non-space characters": a name
- `(.*?), "` is lazy, so it stops at the **first** `, "`, where the description starts
- `"(.*)"$` is greedy, so the description runs to the **last** quote at the end of the line and can contain quotes

Step 3: split the fields, and a small regex for the location

```
parts = fields.split(", ")
for p in parts:
    print(repr(p))
loc = re.fullmatch(r"(.+) from ([\d,-]+)", parts[0])
print(loc.groups())
exons = re.findall(r"(\d+)-(\d+)", loc.group(2))
print(exons)
```

```
'Chr I from 151006-147594,151166-151097'
'Genome Release 64-2-1'
'reverse complement'
'intron sequence removed'
'Verified ORF'
('Chr I', '151006-147594,151166-151097')
[('151006', '147594'), ('151166', '151097')]
```

The first field is the location, the last one is the ORF type, and the strand is minus if one of the fields says `reverse complement`.

Step 4: put it in a function and run it on every header

```
def parse_sgd_header(header):
    """Parse an SGD ORF FASTA header into a dictionary (None if it doesn't fit)."""
    m = header_pat.match(header)
    if not m:
        return None
    orf, gene, sgdid, fields, desc = m.groups()
    parts = fields.split(", ")
    loc = re.fullmatch(r"(.+) from ([\d,-]+)", parts[0])
```

```
    exons = []
    for start, end in re.findall(r"(\d+)-(\d+)", loc.group(2)):
        exons.append((int(start), int(end)))
    strand = "+"
    if "reverse complement" in parts:
        strand = "-"
    return {"orf": orf, "gene": gene, "sgdid": sgdid,
            "chrom": loc.group(1), "strand": strand, "exons": exons,
            "type": parts[-1], "description": desc}

orfs = []
for h in headers:
    info = parse_sgd_header(h)
    if info is None:
        print("could not parse:", h[:60])
    else:
        orfs.append(info)
print(len(orfs), "ORFs parsed")
for key, value in orfs[2].items():
    print(f"{key:12}", str(value)[:65])
```

```
6713 ORFs parsed
orf          YAL003W
gene         EFB1
sgdid        S000000003
chrom        Chr I
strand       +
exons        [(142174, 142253), (142620, 143160)]
type         Verified ORF
description  Translation elongation factor 1 beta; stimulates nucleotide excha
```

Every header parsed. Now we can ask questions:

```
types = Counter()
chroms = Counter()
for info in orfs:
    types[info["type"]] += 1
    chroms[info["chrom"]] += 1
for orf_type, n in types.most_common():
    print(f"{orf_type:26} {n:5}")
print(chroms.most_common(4))

spliced = [info["orf"] for info in orfs if len(info["exons"]) > 1]
print(len(spliced), "ORFs with more than one exon, e.g.", spliced[:3])
```

```
Verified ORF               5111
Dubious ORF                 784
Uncharacterized ORF         709
transposable_element_gene    91
pseudogene                   12
blocked_reading_frame         6
[('Chr IV', 854), ('Chr XV', 607), ('Chr VII', 593), ('Chr XII', 588)]
331 ORFs with more than one exon, e.g. ['YAL001C', 'YAL003W', 'YAL030W']
```

Two things we learned from the data, not from the documentation:

- the location is not always `Chr ...`: the 2-micron plasmid genes say `2-micron plasmid from ...`. A pattern that required `Chr (\S+) from` would have silently skipped them. **Always count how many**

lines your pattern did not match.

- `zcat S_cerevisiae.ORFs.fasta.gz | grep -c -i "verified ORF"` gives 5198, but there are only 5111 Verified ORFs: 87 Dubious ORFs have descriptions such as "partially overlaps verified ORF SEF1/YBL066C". Matching the right field is more reliable than matching anywhere in the line.

Example 2: the GFF3 attribute column

Column 9 of a GFF3 file holds `key=value` pairs separated by `;`:

```
ID=YAL068C;Name=YAL068C;gene=PAU8;Alias=PAU8,seripauperin%20PAU8;
Ontology_term=GO:0003674,GO:0005575,GO:0030437,GO:0045944;Note=Protein%20of...
```

Split or regex?

The format is well defined: pairs are separated by `;` and a key from its value by the first `=`. Plain `split` is enough, and a regex with `findall` works too:

```
attrs = ("ID=YAL068C;Name=YAL068C;gene=PAU8;Alias=PAU8,seripauperin%20PAU8;"
         "Ontology_term=GO:0003674,GO:0005575,GO:0030437,GO:0045944")

def parse_attributes(col9):
    """Turn a GFF3 attribute string into a dictionary."""
    result = {}
    for pair in col9.strip().strip(";").split(";"):
        key, value = pair.split("=", 1)
        result[key] = value
    return result

d1 = parse_attributes(attrs)
d2 = dict(re.findall(r"([^=;]+)=([^;]*)", attrs))
print(d1["gene"], d1["Alias"])
print(d1 == d2)
```

```
PAU8 PAU8,seripauperin%20PAU8
True
```

Where regular expressions help is **inside** the values: getting all the GO IDs out of `Ontology_term`, or pulling one attribute out of a line without parsing everything:

```
print(re.findall(r"GO:\d{7}", attrs))
m = re.search(r"(?:^|;)gene=([^;]+)", attrs)
print(m.group(1))
```

```
['GO:0003674', 'GO:0005575', 'GO:0030437', 'GO:0045944']
PAU8
```

`(?:^|;)gene=` means "`gene=` at the start of the column or right after a `;`". Without it, a search for `ID=` would also match inside a key such as `Parent_ID=`.

GFF3 also encodes special characters: `%20` is a space, `%3B` a `;`, `%2C` a comma. `urllib.parse.unquote()` from the standard library decodes them. (You could write `re.sub` for `%20`, but a library that knows every code is better than a regex that knows three.)

```
from urllib.parse import unquote
print(unquote("Dubious%20open%20reading%20frame%3B%20unlikely%20to%20encode"))
```

```
Dubious open reading frame; unlikely to encode
```

Build a gene table from the GFF3 file

```
genes = {}
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
        attr = parse_attributes(cols[8])
        genes[attr["ID"]] = {
            "chrom": cols[0], "start": int(cols[3]), "end": int(cols[4]),
            "strand": cols[6], "gene": attr.get("gene", attr["ID"]),
            "go": re.findall(r"GO:\d{7}", attr.get("Ontology_term", "")),
            "note": unquote(attr.get("Note", ""))}
print(len(genes), "genes")
for key, value in genes["YAL068C"].items():
    print(f"{key:7}", str(value)[:70])

kinases = [g for g in genes if "GO:0004672" in genes[g]["go"]]
print(len(kinases), "genes annotated with GO:0004672 (protein kinase activity)")
print(sorted(genes[g]["gene"] for g in kinases)[:8])
```

```
6600 genes
chrom   chrI
start   1807
end     2169
strand  -
gene    PAU8
go      ['GO:0003674', 'GO:0005575', 'GO:0030437', 'GO:0045944']
note    Protein of unknown function; member of the seripauperin multigene fami
102 genes annotated with GO:0004672 (protein kinase activity)
['AKL1', 'ALK1', 'ALK2', 'ARK1', 'ATG1', 'BCK1', 'BUB1', 'CDC15']
```

Notice the split of work: `split("\t")` for the columns, `split(";")` for the attributes, and a small regex for the GO IDs. `attr.get(key, default)` handles genes without a `gene=` or `Ontology_term=` attribute. For more complex GFF work, libraries such as `gffutils` or Biopython's GFF support exist.

Example 3: NCBI protein headers, and your own header format

The E. coli protein file uses the older NCBI style header:

```
>gi|388476124|ref|YP_488307.1| thr operon leader peptide [Escherichia coli str. K-12 substr. W3110]
```

The `|` is special in a regex, so it has to be written `\|`. The organism is in `[...]` at the end, which also need escaping:

```
ncbi_pat = re.compile(r"^>gi\|(\d+)\|ref\|([^|]+)\| (.*) \[(.+)\]$")

n_proteins = 0
hypothetical = 0
kinases = []
with gzip.open("E_coli_K12.pep.gz", "rt") as fh:
    for line in fh:
        if not line.startswith(">"):
            continue
        m = ncbi_pat.match(line.rstrip("\n"))
```

```
        if not m:
            print("no match:", line[:60])
            continue
        gi, acc, desc, organism = m.groups()
        n_proteins += 1
        if gi == "388476124":
            print(gi, acc, desc)
            print(organism)
        if re.match(r"hypothetical protein\b", desc):
            hypothetical += 1
        if re.search(r"\bkinase\b", desc):
            kinases.append(acc)
print(n_proteins, "proteins,", hypothetical, "hypothetical,",
      len(kinases), "kinases, e.g.", kinases[:2])
```

```
388476124 YP_488307.1 thr operon leader peptide
Escherichia coli str. K-12 substr. W3110
4213 proteins, 721 hypothetical, 100 kinases, e.g. ['YP_488309.1', 'YP_488331.1']
```

Inside a character class most characters are plain, so `[^|]+` ("anything but a pipe") needs no backslash.

A home-made header

Labs often pack information into headers with their own separators. This header has species, accession and GI separated by `|`, then `KEY=value` fields where the description is quoted:

```
headers = [
    '>Hsapiens|ABC10021.1|gi|1133455 GENE=YFG1 DESC="This gene makes an enzyme"',
    '>Mmusculus|XYZ20002.3|gi|998877 GENE=Yfg1 DESC="Mouse ortholog, putative"',
]
pat = re.compile(r'^>(?P<species>[^|]+)\|(?P<acc>[^|]+)\|gi\|(?P<gi>\d+)'
                 r'\s+GENE=(?P<gene>\S+)\s+DESC="(?P<desc>[^"]*)"')
for h in headers:
    m = pat.match(h)
    if m:
        d = m.groupdict()
        print(d["species"], d["acc"], d["gi"], d["gene"], repr(d["desc"]))
```

```
Hsapiens ABC10021.1 1133455 YFG1 'This gene makes an enzyme'
Mmusculus XYZ20002.3 998877 Yfg1 'Mouse ortholog, putative'
```

Named groups make a long pattern like this much easier to read and to use: `m.groupdict()` is ready to be a row of a table.

Example 4: accession numbers in free text

Methods sections, supplementary tables and README files are full of accession numbers. Each database has a documented format, which translates directly into a regex:

Accession	Format	Pattern
NCBI assembly	`GCF_` or `GCA_`, 9 digits, `.`, version	`\bGC[AF]_\d{9}\.\d+\b`
SRA run	`SRR`, `ERR` or `DRR`, 6 or more digits	`\b[SED]RR\d{6,}\b`
RefSeq protein	`NP_`, `XP_`, `YP_` or `WP_`, digits, version	`\b[NXYW]P_\d+\.\d+\b`
UniProt	see uniprot.org	below
yeast systematic name	e.g. `YAL001C`, `YBL005W-A`	`\bY[A-P][LR]\d{3}[WC](?:-[A-Z])?\b`

```
text = """Reads were deposited as SRR1234567 and SRR1234568 (BioProject
PRJNA123456) and mapped to the S288C assembly GCF_000146045.2 (GenBank
GCA_000146045.2). TFC3 (YAL001C, UniProt P34111) and a paralog of YBL005W-A
were studied. The E. coli protein YP_488307.1 and the human P53 protein
(P04637, TP53_HUMAN) and A0A023GPI8 are controls; SRR12 is a typo."""

patterns = {
    "assembly": r"\bGC[AF]_\d{9}\.\d+\b",
    "SRA run":  r"\b[SED]RR\d{6,}\b",
    "RefSeq":   r"\b[NXYW]P_\d+\.\d+\b",
    "yeast ORF": r"\bY[A-P][LR]\d{3}[WC](?:-[A-Z])?\b",
    "UniProt":  r"\b(?:[OPQ][0-9][A-Z0-9]{3}[0-9]"
                r"|[A-NR-Z][0-9](?:[A-Z][A-Z0-9]{2}[0-9]){1,2})\b",
}
for name, p in patterns.items():
    print(f"{name:10}", re.findall(p, text))
```

```
assembly   ['GCF_000146045.2', 'GCA_000146045.2']
SRA run    ['SRR1234567', 'SRR1234568']
RefSeq     ['YP_488307.1']
yeast ORF  ['YAL001C', 'YBL005W-A']
UniProt    ['P34111', 'P04637', 'A0A023GPI8']
```

- `\b` on both sides stops the pattern from matching inside a longer word: without it the SRA pattern would find `SRR1234567` inside `XSRR1234567`, and `SRR12` is correctly skipped because it has too few digits.
- The UniProt pattern is copied from UniProt's documentation. The two alternatives are wrapped in `(?:...)` so the `\b` applies to both, and so `findall` returns the whole match instead of a group.
- `P53` is a protein name, not an accession, and it is not matched. But no regex can know that a string *is* a real accession; it only checks that it has the right shape.

Example 5: sample and FASTQ file names

Sequencing facilities encode the sample, lane and read number in the file names. Here is a typical folder listing (in real life you'd get this with `os.listdir()` or `glob.glob("*.fastq.gz")`):

```
files = ["sampleA_R1.fastq.gz", "sampleA_R2.fastq.gz",
         "sampleB_R1.fastq.gz", "sampleB_R2.fastq.gz",
         "ctrl-2_S3_L001_R1_001.fastq.gz", "ctrl-2_S3_L001_R2_001.fastq.gz",
         "sampleC_R1.fq.gz", "sampleA_R1.fastq.gz.md5", "notes.txt",
         "Undetermined_S0_L001_R1_001.fastq.gz"]

fastq_pat = re.compile(r"(?P<sample>.+?)(?:_S\d+_L\d{3})?_R(?P<read>[12])"
                       r"(?:_001)?\.f(?:ast)?q\.gz")
for f in files:
    m = fastq_pat.fullmatch(f)
    if m:
        print(f"{f:37} sample={m.group('sample'):12} read={m.group('read')}")
    else:
        print(f"{f:37} skipped")
```

```
sampleA_R1.fastq.gz                   sample=sampleA      read=1
sampleA_R2.fastq.gz                   sample=sampleA      read=2
sampleB_R1.fastq.gz                   sample=sampleB      read=1
sampleB_R2.fastq.gz                   sample=sampleB      read=2
ctrl-2_S3_L001_R1_001.fastq.gz        sample=ctrl-2       read=1
ctrl-2_S3_L001_R2_001.fastq.gz        sample=ctrl-2       read=2
sampleC_R1.fq.gz                      sample=sampleC      read=1
```

```
sampleA_R1.fastq.gz.md5                skipped
notes.txt                              skipped
Undetermined_S0_L001_R1_001.fastq.gz   sample=Undetermined read=1
```

Reading the pattern from left to right:

- `(?P<sample>.+?)` the sample name: anything, as short as possible
- `(?:_S\d+_L\d{3})?` an optional Illumina sample number and lane (`_S3_L001`)
- `_R(?P<read>[12])` the read number, 1 or 2
- `(?:_001)?` an optional `_001`
- `\.f(?:ast)?q\.gz` either `.fastq.gz` or `.fq.gz` (the dots are escaped)

`fullmatch` matters here: with `search`, `sampleA_R1.fastq.gz.md5` would match too. The lazy `.+?` matters too: a greedy `.+` would also work for these names, but lazy is the safer choice when the rest of the pattern is optional.

Now pair the files up by sample and check for missing mates, a common cause of failed pipelines:

```
pairs = {}
for f in files:
    m = fastq_pat.fullmatch(f)
    if not m or m.group("sample") == "Undetermined":
        continue
    sample = m.group("sample")
    if sample not in pairs:
        pairs[sample] = {}
    pairs[sample]["R" + m.group("read")] = f

for sample in sorted(pairs):
    if "R1" in pairs[sample] and "R2" in pairs[sample]:
        print(sample, pairs[sample]["R1"], pairs[sample]["R2"])
    else:
        print(sample, "is missing a mate:", pairs[sample])
```

```
ctrl-2 ctrl-2_S3_L001_R1_001.fastq.gz ctrl-2_S3_L001_R2_001.fastq.gz
sampleA sampleA_R1.fastq.gz sampleA_R2.fastq.gz
sampleB sampleB_R1.fastq.gz sampleB_R2.fastq.gz
sampleC is missing a mate: {'R1': 'sampleC_R1.fq.gz'}
```

This is the same job as the `run_samples.sh` loop in UNIX III, where bash's `${FILE%_R1.fastq.gz}` removed a fixed suffix. The regex also copes with several naming styles at once.

Example 6: cleaning messy table values

Spreadsheets typed by people contain units, thousands separators, stray spaces and many spellings of "missing". Here are values from a "concentration" column and a "chromosome" column:

```
conc = [" 12.5 ug/mL", "3,400", "7", "n/a", "0.8ug/ml", "", "NA", "1.2e3", "~15"]

def clean_number(value):
    """Return the first number in a messy string as a float, or None."""
    value = value.replace(",", "")                  # 3,400 -> 3400
    m = re.search(r"\d+(?:\.\d+)?(?:[eE][-+]?\d+)?", value)
    if m:
        return float(m.group())
    return None

for v in conc:
    print(repr(v), "->", clean_number(v))
```

```
' 12.5 ug/mL' -> 12.5
'3,400' -> 3400.0
'7' -> 7.0
'n/a' -> None
'0.8ug/ml' -> 0.8
'' -> None
'NA' -> None
'1.2e3' -> 1200.0
'~15' -> 15.0
```

- `\d+` the digits before the decimal point
- `(?:\.\d+)?` an optional decimal part (the `.` is escaped)
- `(?:[eE][-+]?\d+)?` an optional exponent such as `e3` or `E-05`

Values like `~15` became `15.0`. Whether that is right is a scientific decision, not a programming one; you may prefer to flag them instead.

Chromosome names are another classic: `chr1`, `Chr01`, `1` and `chromosome_1` all mean the same thing, but they will not join or sort together. Pull out the number and rebuild one standard name:

```python
chroms = ["chr1", "Chr01", "1", "chromosome_1", "CHR10", "chrX", "scaffold_12"]
for c in chroms:
    m = re.fullmatch(r"(?:chr(?:omosome)?_?)?0*(\d+|[XYM])", c, flags=re.I)
    if m:
        print(f"{c:13} -> chr{m.group(1).upper()}")
    else:
        print(f"{c:13} -> not a chromosome name")
```

```
chr1          -> chr1
Chr01         -> chr1
1             -> chr1
chromosome_1  -> chr1
CHR10         -> chr10
chrX          -> chrX
scaffold_12   -> not a chromosome name
```

`0*` drops leading zeros, so `Chr01` and `chr1` both become `chr1`.

Coordinate ranges

Coordinates are written in many styles: `10..30` (GenBank), `10-30`, `chrI:10-30` and `complement(3300..4037)`. `re.split` handles several separators at once, and groups pull a range apart:

```python
text = ["ABC\t10..30",
        "ABC   30..40",
        "DEF 55-70"]
for row in text:
    # split on a tab or a run of spaces, OR on ".." OR on "-"
    name, start, end = re.split(r"\s+|\.\.|-", row)
    print(name, int(start), int(end))

locations = ["190..255", "complement(3300..4037)", "join(337..2799,2801..3733)"]
for loc in locations:
    strand = "-" if loc.startswith("complement") else "+"
    ranges = [(int(s), int(e)) for s, e in re.findall(r"(\d+)\.\.(\d+)", loc)]
    print(f"{loc:28} strand {strand} {ranges}")
```

```
ABC 10 30
ABC 30 40
```

```
DEF 55 70
190..255                    strand + [(190, 255)]
complement(3300..4037)      strand - [(3300, 4037)]
join(337..2799,2801..3733)  strand + [(337, 2799), (2801, 3733)]
```

(GenBank features are best read with Biopython, which understands all location types; see Python IV.)

Example 7: pandas `.str.extract`

If your data is already in a pandas table (Pandas), you do not need a loop: the `.str` methods apply a regex to a whole column. `.str.extract()` makes one new column per group, named after the named groups:

```
import pandas as pd

df = pd.DataFrame({"file": ["sampleA_R1.fastq.gz", "sampleA_R2.fastq.gz",
                            "ctrl-2_S3_L001_R1_001.fastq.gz", "notes.txt"]})
parts = df["file"].str.extract(r"^(?P<sample>.+?)(?:_S\d+_L\d{3})?"
                               r"_R(?P<read>[12])(?:_001)?\.f(?:ast)?q\.gz$")
df = pd.concat([df, parts], axis=1)
print(df)
print(df["file"].str.contains(r"_R1[._]").sum(), "R1 files")
```

```
                           file   sample read
0            sampleA_R1.fastq.gz  sampleA    1
1            sampleA_R2.fastq.gz  sampleA    2
2  ctrl-2_S3_L001_R1_001.fastq.gz   ctrl-2    1
3                     notes.txt      NaN  NaN
2 R1 files
```

Rows that do not match get `NaN` (missing). Related methods are `.str.contains()` (true/false per row), `.str.replace(p, r, regex=True)` and `.str.findall()`.

Exercise solutions

These are one way to solve the Python V exercises. They reuse `read_fasta()`, `find_motif()` and `revcomp()` from Python V, repeated here so this section runs on its own:

```
def read_fasta(filename):
    """Read a gzipped FASTA file into a dictionary: {id: sequence}."""
    seqs = {}
    seq_id = None
    with gzip.open(filename, "rt") as fh:
        for line in fh:
            line = line.strip()
            if line.startswith(">"):
                seq_id = line[1:].split()[0]
                seqs[seq_id] = []
            else:
                seqs[seq_id].append(line)
    for seq_id in seqs:
        seqs[seq_id] = "".join(seqs[seq_id])
    return seqs

IUPAC = {"A": "A", "C": "C", "G": "G", "T": "T",
         "R": "[AG]", "Y": "[CT]", "S": "[CG]", "W": "[AT]",
         "K": "[GT]", "M": "[AC]", "B": "[CGT]", "D": "[AGT]",
         "H": "[ACT]", "V": "[ACG]", "N": "[ACGT]"}
```

```python
def iupac_to_regex(motif):
    pattern = ""
    for base in motif.upper():
        pattern += IUPAC[base]
    return pattern

def revcomp(seq):
    comp = {"A": "T", "C": "G", "G": "C", "T": "A", "R": "Y", "Y": "R",
            "S": "S", "W": "W", "K": "M", "M": "K", "B": "V", "V": "B",
            "D": "H", "H": "D", "N": "N"}
    rc = ""
    for base in reversed(seq.upper()):
        rc += comp[base]
    return rc

def find_motif(motif, seq):
    hits = []
    for m in re.finditer(iupac_to_regex(motif), seq):
        hits.append((m.start() + 1, "+"))
    rc_motif = revcomp(motif)
    if rc_motif != motif:
        for m in re.finditer(iupac_to_regex(rc_motif), seq):
            hits.append((m.start() + 1, "-"))
    return sorted(hits)

genome = read_fasta("S_cerevisiae.fasta.gz")
```

1. Validate sequences

```python
def is_dna(seq):
    return re.fullmatch(r"[ACGT]+", seq, flags=re.IGNORECASE) is not None

def is_protein(seq):
    return re.fullmatch(r"[ACDEFGHIKLMNPQRSTVWY]+", seq) is not None

for s in ["ACGTN", "acgt", "MKRISTT", "MKRIS*"]:
    print(f"{s:8} dna={is_dna(s)} protein={is_protein(s)}")
```

```
ACGTN    dna=False protein=True
acgt     dna=True protein=False
MKRISTT  dna=False protein=True
MKRIS*   dna=False protein=False
```

`fullmatch` (not `search`) makes sure *every* character is allowed. Note that `ACGT` sequences also pass `is_protein`, because A, C, G and T are amino acid letters too. Real protein files also contain `*` (stop) and `X` (unknown); add them to the class if you want to accept them.

2. EcoRI sites per chromosome

```python
ecoRI = re.compile(r"GAATTC")
print(f"{'chrom':7} {'length':>9} {'sites':>6} {'per 10kb':>9}")
for name, seq in genome.items():
    n = len(ecoRI.findall(seq))
    print(f"{name:7} {len(seq):9} {n:6} {n / len(seq) * 10000:9.2f}")
```

```
chrom      length  sites  per 10kb
chrI       230218     79      3.43
```

```
chrII      813184    283      3.48
chrIII     316620    103      3.25
chrIV     1531933    576      3.76
chrV       576874    184      3.19
chrVI      270161     85      3.15
chrVII    1090940    392      3.59
chrVIII    562643    179      3.18
chrIX      439888    150      3.41
chrX       745751    280      3.75
chrXI      666816    251      3.76
chrXII    1078177    422      3.91
chrXIII    924431    346      3.74
chrXIV     784333    312      3.98
chrXV     1091291    392      3.59
chrXVI     948066    334      3.52
chrmt       85779     10      1.17
```

EcoRI is a palindrome, so this counts both strands. The expected rate for a random 6-base site is 10000 / 4096 = 2.4 per 10 kb.

3. Sample names

```
files = ["sampleA_R1.fastq.gz", "sampleA_R2.fastq.gz", "ctrl-2_R1.fq.gz",
         "sampleA_R1.fastq.gz.md5", "notes.txt"]
for f in files:
    m = re.fullmatch(r"(.+)_R([12])\.f(?:ast)?q\.gz", f)
    if m:
        print(m.group(1), "read", m.group(2))
```

```
sampleA read 1
sampleA read 2
ctrl-2 read 1
```

4. Accessions

```
text = ("Reads SRR1234567 and ERR998877 were mapped to GCF_000146045.2 "
        "(R64); see also GCA_000146045.2 and SRR12.")
print(re.findall(r"\bGCF_\d{9}\.\d+\b", text))
print(re.findall(r"\b[SED]RR\d{6,}\b", text))
```

```
['GCF_000146045.2']
['SRR1234567', 'ERR998877']
```

5. SCB sites on chromosome III, both strands

```
print(revcomp("CRCGAAA"), "palindromic:", revcomp("CRCGAAA") == "CRCGAAA")
hits = find_motif("CRCGAAA", genome["chrIII"])
strands = Counter(strand for pos, strand in hits)
print(len(hits), "sites:", dict(strands))
print(hits[:4])
```

```
TTTCGYG palindromic: False
61 sites: {'+': 42, '-': 19}
[(7239, '+'), (17160, '+'), (18097, '+'), (29179, '-')]
```

6. Overlapping motifs

```
print("TATATATA".count("TATA"),
      len(re.findall(r"TATA", "TATATATA")),
```

```
      len(re.findall(r"(?=(TATA))", "TATATATA")))

tata = iupac_to_regex("TATAWAWR")
print(tata)
chrI = genome["chrI"]
print(len(re.findall(tata, chrI)), "non-overlapping,",
      len(re.findall("(?=(" + tata + "))", chrI)), "overlapping")
```

```
2 2 3
TATA[AT]A[AT][AG]
102 non-overlapping, 132 overlapping
```

`TATAWAWR` can overlap itself (for example in `TATATATAAG`), so the two counts differ. This counts the forward strand only; use `find_motif()`-style logic with the lookahead to add the minus strand.

7. Coordinates with `re.sub` and groups

```
loc = "Chr IV from 1802-2953"
region = re.sub(r"^Chr (\S+) from (\d+)-(\d+)$", r"chr\1:\2-\3", loc)
print(region)

m = re.fullmatch(r"(\w+):(\d+)-(\d+)", region)
coords = (m.group(1), int(m.group(2)), int(m.group(3)))
print(coords)
```

```
chrIV:1802-2953
('chrIV', 1802, 2953)
```

8. N-glycosylation motifs in E. coli proteins

`{P}` in PROSITE means "not P", which is `[^P]`. Sites can overlap (as in `NNSTS`), so use a lookahead:

```
proteins = read_fasta("E_coli_K12.pep.gz")
glyc = re.compile(r"(?=(N[^P][ST][^P]))")

with_site = 0
total_sites = 0
for name, seq in proteins.items():
    n = len(glyc.findall(seq))
    total_sites += n
    if n > 0:
        with_site += 1
print(with_site, "of", len(proteins), "proteins have at least one site;",
      total_sites, "sites in total")
print(glyc.findall("MNNSTSAQ"))
```

```
2557 of 4213 proteins have at least one site; 5408 sites in total
['NNST', 'NSTS']
```

(E. coli does not N-glycosylate its own proteins, so most of these are just chance matches of a short, common motif - a good reminder that a pattern match is not a biological function.)

See also

- Python V: Regular Expressions for the syntax and a quick-reference table
- regex101.com (choose the Python flavor) to test and explain patterns
- Python IV: Packages for Biopython parsers
- Pandas and Plotting for analysing and plotting what you extracted