Python V: Regular Expressions

A **regular expression** (regex) is a small language for describing a *pattern* of text: "GAATTC", but also "GT, then C or T, then A or G, then AC", "one or more digits", or "everything between `ID=` and the next `;`". Python's `re` module tests whether a string contains a pattern, finds where it is, pulls out pieces of it, and replaces or splits on it.

You have already used patterns: `grep -E` and `sed` in the UNIX III lab use regular expressions, and almost all of that syntax works the same way in Python.

What you'll learn

- when plain string methods (`in`, `find`, `split`, `replace`) are enough, and when you need a regex
- the `re` functions: `re.search`, `re.match`, `re.fullmatch`, `re.findall`, `re.finditer`, `re.sub`, `re.split`, `re.compile`, and flags such as `re.IGNORECASE`
- why patterns are written as raw strings, `r"..."`
- the syntax, one piece at a time: characters, `.`, character classes like `[ACGT]`, `\d \w \s`, anchors `^ $ \b`, quantifiers `* + ? {n,m}`, greedy and lazy matching, alternation `|`, groups `( )` and named groups
- how to find overlapping matches (for example overlapping motifs)
- how to use all of this on real data: restriction sites on both DNA strands, degenerate (IUPAC) motifs, start/stop codon patterns and FASTA headers

Worked examples on real files (FASTA headers, GFF3 attributes, file names, accession numbers, messy tables) and the solutions to the exercises are in Regular Expression Worked Examples.

Setup

The examples build on each other, so run them in order in one Jupyter notebook (or one `python3` session). The later examples use the yeast genome. On the cluster, make a folder and download the data:

```
mkdir -p ~/bigdata/gen220/python5
cd ~/bigdata/gen220/python5
URL=https://github.com/biodataprog/GEN220_data/raw/main/genome
curl -L -O $URL/S_cerevisiae.fasta.gz
curl -L -O $URL/S_cerevisiae.ORFs.fasta.gz
curl -L -O $URL/E_coli_K12.pep.gz
```

1. Do you need a regular expression?

Start with the string methods from Python I. They are simple, fast and easy to read, and they are all you need when you are looking for one **exact** piece of text:

```
seq = "ACAGACGAGAGAATTCGGTAGATGAATTCA"

print("GAATTC" in seq)            # is EcoRI's site in the sequence?
print(seq.find("GAATTC"))         # position of the first site (-1 if none)
print(seq.count("GAATTC"))        # how many sites
print(seq.startswith("ACAG"), seq.endswith("TTCA"))

header = "YAL001C TFC3 SGDID:S000000001, Chr I from 151006-147594"
print(header.split()[0])          # the ORF name
print(header.replace("Chr I", "chrI"))
```

```
True
10
2
True True
YAL001C
YAL001C TFC3 SGDID:S000000001, chrI from 151006-147594
```

Now think about these tasks:

- Find **HincII** sites. HincII cuts `GTYRAC`, where `Y` is C or T and `R` is A or G. That is four different sequences (GTCAAC, GTCGAC, GTTAAC, GTTGAC). With string methods you need four searches.
- Pull out every RefSeq assembly accession (`GCF_` then some digits, a dot and a version number) from a paragraph of text.
- Check that a sequence contains only A, C, G and T.
- Find every in-frame stretch from `ATG` to a stop codon.

None of these is one exact string. They are **patterns**, and that is what regular expressions are for.

Rule of thumb: if you can say what you want as one exact string, use a string method. If you catch yourself writing "any digit", "one or more", "either this or that", or "the text between X and Y", use a regex.

2. A first regular expression: `re.search`

```
import re

seq = "ACAGACGAGAGAATTCGGTAGATGAATTCA"
m = re.search(r"GAATTC", seq)
print(m)
```

```
<re.Match object; span=(10, 16), match='GAATTC'>
```

`re.search(pattern, string)` scans the string from left to right and stops at the **first** place the pattern matches. It returns a **match object**, or `None` if there is no match. Because `None` counts as false, the usual way to use it is with `if`:

```
for site in [r"GAATTC", r"GGATCC"]:          # EcoRI, BamHI
    m = re.search(site, seq)
    if m:
        print(site, "found at", m.start())
    else:
        print(site, "not found")
```

```
GAATTC found at 10
GGATCC not found
```

The match object tells you what matched and where:

Method	Returns
`m.group()` or `m.group(0)`	the text that matched
`m.start()`	position where the match starts
`m.end()`	position just **after** the match ends
`m.span()`	`(start, end)` as a tuple

```
m = re.search(r"GAATTC", seq)
print(m.group(), m.start(), m.end(), m.span())
print(seq[m.start():m.end()])
```

```
GAATTC 10 16 (10, 16)
GAATTC
```

The positions work exactly like string slicing: they start at 0 and the end is not included, so `seq[m.start():m.end()]` is the match. Genome coordinates in BED files are also 0-based like this, but GFF3, VCF and most genome browsers count from 1, so the site above starts at base `m.start() + 1` = 11 in GFF coordinates.

If there is no match, `re.search` returns `None`, and calling `.group()` on `None` fails. Always test the result before you use it:

```
m = re.search(r"GGATCC", seq)
print(m)
try:
    print(m.group())
except AttributeError as err:
    print("Error:", err)
```

```
None
Error: 'NoneType' object has no attribute 'group'
```

3. Raw strings: always write patterns as `r"..."`

In a normal Python string the backslash starts an *escape sequence*: `"\n"` is a newline and `"\t"` is a tab. Regular expressions also use the backslash: `\d` means "a digit" and `\b` means "a word boundary". The two meanings clash:

```
print(len("\t"), len(r"\t"))      # a tab character vs backslash + t
print(len("\b"), len(r"\b"))      # "\b" is a backspace character in Python!

print(re.search("\bEcoRI\b", "cut with EcoRI today"))
print(re.search(r"\bEcoRI\b", "cut with EcoRI today"))
```

```
1 2
1 2
None
<re.Match object; span=(9, 14), match='EcoRI'>
```

Without the `r`, Python turned `\b` into a backspace character before `re` ever saw it, so the pattern silently matched nothing. A **raw string** (`r"..."`) keeps every backslash as it is and hands it to `re` unchanged. Python 3.12 and newer also warn about unknown escapes such as `"\d"` (`SyntaxWarning: invalid escape sequence`). Make it a habit: **every regex is a raw string**.

4. Building patterns: characters and character classes

Letters, digits and special characters

Most characters match themselves: the pattern `GAATTC` matches the text `GAATTC`. Matching is case-sensitive, so `GAATTC` does not match `gaattc` (see flags in section 10). These characters have special meanings:

```
.  ^  $  *  +  ?  {  }  [  ]  \  |  (  )
```

To match one of them literally, put a backslash in front of it. For example `\.` matches a real dot and `\|` a real pipe. `re.escape()` adds the backslashes for you, which is useful when a pattern comes from data:

```
print(re.search(r"YP_488307.1", "ref|YP_488307x1|"))   # . matches anything
print(re.search(r"YP_488307\.1", "ref|YP_488307x1|"))  # \. only matches a dot
print(re.escape("YP_488307.1"))
```

```
<re.Match object; span=(4, 15), match='YP_488307x1'>
None
YP_488307\.1
```

`.` matches any single character

`.` matches any one character except a newline:

```
for codon in ["TAA", "TGA", "TAG", "TTA"]:
    if re.search(r"T.A", codon):
        print(codon, "matches T.A")
```

```
TAA matches T.A
TGA matches T.A
TTA matches T.A
```

Character classes: `[...]`

Square brackets match **one** character from a set. `[AG]` is "A or G" (a purine), `[CT]` is "C or T" (a pyrimidine), `[ACGT]` is any base:

```
hincII = r"GT[CT][AG]AC"          # GTYRAC
dna = "AAGTCAACTTGTTGACGGGTCGACAAGTACAC"
for m in re.finditer(hincII, dna):
    print(m.group(), m.start())
```

```
GTCAAC 2
GTTGAC 10
GTCGAC 18
```

(`re.finditer` gives every match, one after another; section 8 explains it.)

Inside brackets:

- a **range** uses `-`: `[0-9]` is any digit, `[A-Z]` any capital letter, `[A-Za-z]` any letter
- a `^` right after `[` **negates** the class: `[^ACGT]` is any character that is *not* A, C, G or T, and `[^;]` is anything except a semicolon
- most special characters lose their meaning, so `[.]` is a plain dot

```
print(re.findall(r"[^ACGT]", "ACGTNNACGRTX"))   # characters that are not bases
print(re.findall(r"[0-9]", "chr12:1500"))
```

```
['N', 'N', 'R', 'X']
['1', '2', '1', '5', '0', '0']
```

Shorthand classes

Some classes are used so often that they have short names:

Shorthand	Means	Same as
`\d`	a digit	`[0-9]`
`\w`	a "word" character: letter, digit or underscore	`[A-Za-z0-9_]`
`\s`	whitespace: space, tab, newline	`[ \t\n\r\f\v]`
`\D`, `\W`, `\S`	the opposite: NOT a digit / word char / whitespace	`[^0-9]` etc.

```
line = "chrI\tSGD\tgene\t335\t649"
print(re.findall(r"\d", line))
print(re.findall(r"\s", line))
print(re.findall(r"\S", "Chr I"))
```

```
['3', '3', '5', '6', '4', '9']
['\t', '\t', '\t', '\t']
['C', 'h', 'r', 'I']
```

Each of these matches **one** character. To match a whole number you need a quantifier, which is next.

5. Quantifiers: how many times?

A quantifier goes right after a character, class or group and says how many times it may repeat:

Quantifier	Means
`*`	0 or more times
`+`	1 or more times
`?`	0 or 1 time (optional)
`{n}`	exactly n times
`{n,m}`	between n and m times
`{n,}`	n or more times

```
line = "chrI\tSGD\tgene\t335\t649"
print(re.findall(r"\d+", line))            # whole numbers now

print(re.search(r"\d+ reads?", "1 read").group())
print(re.search(r"\d+ reads?", "250 reads").group())

print(re.findall(r"A{4,}", "GCAAAATTTAAAAAAAGCAAG"))   # runs of 4 or more A
print(re.findall(r"(?:CA){3,}", "TTCACACACAGGCATTCACACACACA"))  # CA repeats
```

```
['335', '649']
1 read
250 reads
['AAAA', 'AAAAAAA']
['CACACACA', 'CACACACACA']
```

(`(?:CA)` groups `CA` so the `{3,}` applies to both letters; see section 7.)

Greedy and lazy matching

Quantifiers are **greedy**: they match as much as possible. Adding `?` after a quantifier makes it **lazy**: it matches as little as possible. This matters most with `.*` ("anything"):

```
attrs = "ID=YAL069W;Name=YAL069W;Ontology_term=GO:0003674"
print(re.search(r"ID=(.*);", attrs).group(1))     # greedy: up to the LAST ;
print(re.search(r"ID=(.*?);", attrs).group(1))    # lazy: up to the FIRST ;
print(re.search(r"ID=([^;]+)", attrs).group(1))   # "not a ;" - often clearest
```

```
YAL069W;Name=YAL069W
YAL069W
YAL069W
```

The greedy `.*` ran all the way to the last `;`. Both fixes work; a negated class such as `[^;]+` ("one or more characters that are not `;`") says exactly what you mean and is usually the best choice. `( )` captures a piece of the match as `group(1)`; see section 7.)

6. Anchors, and `match` vs `fullmatch`

Anchors match a *position*, not a character:

Anchor	Matches
`^`	the start of the string
`$`	the end of the string (or just before a final newline)
`\b`	a word boundary: between a `\w` character and a non-`\w` character

```
print(re.search(r"^ATG", "ATGAAACCCTAA") is not None)   # starts with ATG?
print(re.search(r"^ATG", "CCATGAAATAA") is not None)
print(re.search(r"TAA$", "ATGAAACCCTAA") is not None)   # ends with TAA?
```

```
print(re.findall(r"Chr1", "Chr1 Chr10 Chr11"))
print(re.findall(r"\bChr1\b", "Chr1 Chr10 Chr11"))     # like grep -w
```

```
True
False
True
['Chr1', 'Chr1', 'Chr1']
['Chr1']
```

`\bChr1\b` does the same job as `grep -w Chr1`: it will not match `Chr10`.

The `re` module has three ways to match, depending on how much of the string must match:

Function	The pattern must match...
`re.search(p, s)`	anywhere in `s`
`re.match(p, s)`	at the **start** of `s` (like adding `^`)
`re.fullmatch(p, s)`	the **whole** of `s` (like adding `^` and `$`)

`re.fullmatch` is the right tool to **validate** data. Is this a DNA sequence?

```
for s in ["ACGTTGCA", "ACGTNNGCA", "acgt", ""]:
    print(repr(s),
          re.search(r"[ACGT]+", s) is not None,
          re.match(r"[ACGT]+", s) is not None,
          re.fullmatch(r"[ACGT]+", s) is not None)
```

```
'ACGTTGCA' True True True
'ACGTNNGCA' True True False
'acgt' False False False
'' False False False
```

Only `fullmatch` rejects `ACGTNNGCA`: `search` and `match` are happy as soon as *some* part of the string is made of bases.

7. Alternation and groups

Alternation: `|` means "or"

```
stops = r"TAA|TAG|TGA"
print(re.findall(stops, "ATGTAAGGTAGCCTGA"))
```

```
['TAA', 'TAG', 'TGA']
```

`|` has the lowest priority of anything in a pattern, so `ATG|GTG` means "`ATG` or `GTG`", and `chrI|II` means "`chrI` or `II`" - not `chrI` or `chrII`. Use parentheses to limit it: `chr(I|II)`.

Groups: `( )`

Parentheses do two jobs:

1. they **group** a piece of pattern so a quantifier or `|` applies to all of it, as in `(CA){3,}` or `chr(I|II)`
2. they **capture** the text that the piece matched, so you can get it back with `m.group(1)`, `m.group(2)` ... (numbered by counting opening parentheses from the left). `m.groups()` returns all of them as a tuple.

Here we take apart a yeast ORF FASTA header:

```
header = (">YAL001C TFC3 SGDID:S000000001, Chr I from 151006-147594,"
          "151166-151097, Genome Release 64-2-1, reverse complement")
m = re.search(r"^>(\S+) (\S+) SGDID:(S\d+), Chr (\S+) from (\d+)-(\d+)", header)
```

```
if m:
    print(m.groups())
    print("ORF:", m.group(1), " gene:", m.group(2), " chrom:", m.group(4))
    start, end = int(m.group(5)), int(m.group(6))
    print("first exon length:", abs(end - start) + 1)
```

```
('YAL001C', 'TFC3', 'S000000001', 'I', '151006', '147594')
ORF: YAL001C  gene: TFC3  chrom: I
first exon length: 3413
```

Captured groups are always **strings**; convert numbers with `int()` before doing arithmetic.

Named groups: `(?P<name>...)`

With many groups, numbers get confusing. Give each group a name with `(?P<name>...)` and read it with `m.group("name")`, or get all of them as a dictionary with `m.groupdict()`:

```
pat = (r"^>(?P<orf>\S+) (?P<gene>\S+) SGDID:(?P<sgdid>S\d+), "
       r"Chr (?P<chrom>\S+) from (?P<start>\d+)-(?P<end>\d+)")
m = re.search(pat, header)
print(m.group("gene"))
info = m.groupdict()
print(type(info), len(info), "groups")
for key, value in info.items():
    print(f"  {key:6} = {value}")
```

```
TFC3
<class 'dict'> 6 groups
  orf    = YAL001C
  gene   = TFC3
  sgdid  = S000000001
  chrom  = I
  start  = 151006
  end    = 147594
```

(A long pattern can be split over several lines: Python joins string literals that sit next to each other inside parentheses.)

Non-capturing groups: `(?:...)`

Sometimes you need parentheses only to group, not to capture. `(?:...)` groups without creating a numbered group. This matters most with `re.findall` (next section), which returns the groups instead of the whole match when a pattern has groups:

```
text = "chrI chrII chrV chrIX chrmt"
print(re.findall(r"chr(I|II)\b", text))      # returns only the group
print(re.findall(r"chr(?:I|II)\b", text))    # returns the whole match
```

```
['I', 'II']
['chrI', 'chrII']
```

8. Finding every match: `findall` and `finditer`

`re.search` stops at the first match. To get all of them:

- `re.findall(p, s)` returns a **list of strings**: the matches, or the groups if the pattern has groups (a list of tuples if it has more than one group)
- `re.finditer(p, s)` returns match objects one at a time, so you also get the **positions**

```
seq = "ATGGAATTCAAGGATCCTTGAATTCGGATCCAAGCTT"
print(re.findall(r"GAATTC", seq))
print(len(re.findall(r"GAATTC", seq)), "EcoRI sites")

for m in re.finditer(r"GAATTC|GGATCC", seq):
    print(m.group(), "at", m.start() + 1, "-", m.end())    # 1-based, like GFF
```

```
['GAATTC', 'GAATTC']
2 EcoRI sites
GAATTC at 4 - 9
GGATCC at 12 - 17
GAATTC at 20 - 25
GGATCC at 26 - 31
```

With more than one group, `findall` gives tuples, which fit nicely into a dictionary:

```
attrs = "ID=YAL069W;Name=YAL069W;orf_classification=Dubious"
pairs = re.findall(r"([^=;]+)=([^;]*)", attrs)
print(pairs)
print(dict(pairs))
```

```
[('ID', 'YAL069W'), ('Name', 'YAL069W'), ('orf_classification', 'Dubious')]
{'ID': 'YAL069W', 'Name': 'YAL069W', 'orf_classification': 'Dubious'}
```

Overlapping matches

Both `findall` and `finditer` continue searching **after** the end of the previous match, so matches never overlap. That is usually what you want for restriction sites, but not always for motifs:

```
dna = "ATATATAT"
print(dna.count("ATA"))                 # string count: non-overlapping
print(re.findall(r"ATA", dna))          # also non-overlapping
print(re.findall(r"(?=(ATA))", dna))    # overlapping
for m in re.finditer(r"(?=(ATA))", dna):
    print(m.start(), m.group(1))
```

```
2
['ATA', 'ATA']
['ATA', 'ATA', 'ATA']
0 ATA
2 ATA
4 ATA
```

`(?=...)` is a **lookahead**: it checks that the pattern matches starting at this position, but does not move forward or "use up" any characters. So the search tries every position, one base at a time. Put a capturing group inside it, `(?=(...))`, to get the text back. You don't need other lookaround tricks for this class; just remember this recipe for overlapping matches.

9. Replacing and splitting: `re.sub` and `re.split`

`re.sub(pattern, replacement, string)`

`re.sub` replaces **every** match (use `count=1` to replace only the first):

```
print(re.sub(r"\s+", " ", "too    many\t\tspaces  here"))
print(re.sub(r"[^ACGT]", "N", "ACGTRYKMACGT"))   # mask non-ACGT characters
print(re.sub(r"Chr", "chr", "Chr1 Chr2 Chr3", count=1))
```

```
too many spaces here
ACGTNNNNACGT
chr1 Chr2 Chr3
```

The replacement can use the groups from the match: `\1`, `\2` ... or `\g<name>` for named groups (write the replacement as a raw string too). Here we turn SGD's `Chr I from 151006-147594` style into `chrI:151006-147594`, and swap `gene_ORF` into `ORF_gene`:

```
text = "Chr I from 151006-147594"
print(re.sub(r"Chr (\S+) from (\d+)-(\d+)", r"chr\1:\2-\3", text))

print(re.sub(r"(?P<gene>\w+)_(?P<orf>Y[A-P][LR]\d{3}[WC])",
             r"\g<orf>_\g<gene>", "TFC3_YAL001C VPS8_YAL002W"))
```

```
chrI:151006-147594
YAL001C_TFC3 YAL002W_VPS8
```

`re.split(pattern, string)`

`str.split` splits on one fixed separator. `re.split` splits on a pattern:

```
print("ABC  10..30".split(" "))            # empty string from double space
print(re.split(r"\s+", "ABC  10..30"))     # any run of whitespace
print(re.split(r"\s+|\.\.", "ABC\t10..30"))  # whitespace OR ".."
print(re.split(r"[,;]\s*", "YAL001C, YAL002W;YAL003W ,YAL004W"))
```

```
['ABC', '', '10..30']
['ABC', '10..30']
['ABC', '10', '30']
['YAL001C', 'YAL002W', 'YAL003W ', 'YAL004W']
```

10. Compiling patterns and flags

`re.compile(pattern)` turns a pattern into a **pattern object** with the same methods (`.search`, `.findall`, `.finditer`, `.sub` ...). Python caches recent patterns, so compiling is not much faster, but it lets you give a pattern a name and define it once, outside a loop:

```
ecoRI = re.compile(r"GAATTC")
for s in ["GGAATTCC", "GGATCC", "GAATTCGAATTC"]:
    print(s, len(ecoRI.findall(s)))
```

```
GGAATTCC 1
GGATCC 0
GAATTCGAATTC 2
```

Flags change how a pattern matches. The most useful one is `re.IGNORECASE` (short name `re.I`); many genome files use lowercase letters for repeat-masked (soft-masked) sequence:

```
softmasked = "ACGTgaattcACGTGAATTC"
print(re.findall(r"GAATTC", softmasked))
print(re.findall(r"GAATTC", softmasked, flags=re.IGNORECASE))
pat = re.compile(r"GAATTC", re.IGNORECASE)
print(pat.findall(softmasked))
```

```
['GAATTC']
['gaattc', 'GAATTC']
['gaattc', 'GAATTC']
```

Other flags: `re.MULTILINE` (`re.M`) makes `^` and `$` match at the start and end of every line in a multi-line string, and `re.VERBOSE` (`re.X`) lets you spread a pattern over several lines with comments. Combine flags with `|`, e.g. `flags=re.I | re.M`.

Pass flags by name. The third argument of `re.sub` is `count`, not `flags`, and the second argument of a compiled pattern's `.search` is the **start position**. Passing a flag in the wrong place does not raise an error, it just silently does something else:

```
print(re.sub(r"gaattc", "NNNNNN", softmasked, re.I))        # WRONG: count=2
print(re.sub(r"gaattc", "NNNNNN", softmasked, flags=re.I))  # right
print(re.compile(r"ACGT").search(softmasked, re.I))         # WRONG: pos=2
```

```
ACGTNNNNNNACGTGAATTC
ACGTNNNNNNACGTNNNNNN
<re.Match object; span=(10, 14), match='ACGT'>
```

`re.I` is really the number 2. The first line replaced up to 2 matches, case-sensitively, so only the lowercase site changed. The last line started searching at position 2 and skipped the `ACGT` at position 0. Python 3.13 and newer print a `DeprecationWarning` for the first line.

11. Putting it together on the yeast genome

First read the yeast genome into a dictionary (the `gzip` module opens `.gz` files directly; see Python IV):

```
import gzip

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

genome = read_fasta("S_cerevisiae.fasta.gz")
chrI = genome["chrI"]
print(len(genome), "sequences; chrI is", len(chrI), "bp")
```

```
17 sequences; chrI is 230218 bp
```

Restriction sites on chromosome I

A 6-base site is expected about once every 4^6 = 4096 bases in random DNA. How does chromosome I compare?

```
enzymes = {"EcoRI": r"GAATTC", "BamHI": r"GGATCC", "HindIII": r"AAGCTT",
           "NotI": r"GCGGCCGC", "HincII": r"GT[CT][AG]AC"}
for name, site in enzymes.items():
    n = len(re.findall(site, chrI))
    print(f"{name:8} {site:14} {n:4} sites")
print("expected for a 6-cutter:", round(len(chrI) / 4**6))
```

```
EcoRI    GAATTC           79 sites
BamHI    GGATCC           22 sites
HindIII  AAGCTT           64 sites
NotI     GCGGCCGC          1 sites
HincII   GT[CT][AG]AC    168 sites
expected for a 6-cutter: 56
```

Yeast DNA is AT-rich (about 38% GC), so the GC-rich 8-base NotI site is rare. HincII finds more sites because its degenerate site matches four sequences.

Degenerate sites: turning IUPAC codes into a regex

Motifs and enzyme sites are often written with IUPAC codes: `R` = A/G, `Y` = C/T, `N` = any base, and so on. Each code is a character class, so a dictionary can translate a motif into a regex:

```
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]"}

def iupac_to_regex(motif):
    """Convert an IUPAC DNA motif such as GTYRAC into a regex."""
    pattern = ""
    for base in motif.upper():
        pattern += IUPAC[base]
    return pattern

print(iupac_to_regex("GTYRAC"))      # HincII
print(iupac_to_regex("TGASTCA"))     # Gcn4 binding site: TGA(C/G)TCA
```

```
GT[CT][AG]AC
TGA[CG]TCA
```

Both strands

A motif can be on either strand of the DNA. Instead of searching a second, reverse-complemented copy of the chromosome, search the forward strand for the motif **and** for its reverse complement. Then every position is on the same (forward strand) coordinates:

```
def revcomp(seq):
    """Reverse complement of a DNA sequence (IUPAC codes included)."""
    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):
    """Return a list of (start, strand) for an IUPAC motif; start is 1-based."""
    hits = []
    for m in re.finditer(iupac_to_regex(motif), seq):
        hits.append((m.start() + 1, "+"))
    rc_motif = revcomp(motif)
    if rc_motif != motif:          # palindromes would be counted twice
        for m in re.finditer(iupac_to_regex(rc_motif), seq):
            hits.append((m.start() + 1, "-"))
    return sorted(hits)

print(revcomp("GAATTC"), revcomp("TGASTCA"), revcomp("TGAAACA"))
for motif in ["GAATTC", "TGASTCA", "TGAAACA"]:
    hits = find_motif(motif, chrI)
    print(motif, len(hits), "hits; first three:", hits[:3])
```

```
GAATTC TGASTCA TGTTTCA
GAATTC 79 hits; first three: [(2611, '+'), (2662, '+'), (6852, '+')]
TGASTCA 19 hits; first three: [(18326, '+'), (19489, '+'), (22218, '+')]
TGAAACA 64 hits; first three: [(3598, '+'), (4006, '-'), (6887, '+')]
```

Most restriction sites (EcoRI, BamHI, HincII) are **palindromes**: the site is its own reverse complement, so one search already covers both strands, and searching the reverse complement too would count every site twice. Some transcription factor sites are palindromic too (Gcn4, `TGASTCA`), but many are not: the pheromone response element `TGAAACA`, bound by Ste12, reads `TGTTTCA` on the forward strand when it sits on the minus strand, and about half of its hits are found only by the reverse-complement search.

Start and stop codons: a simple ORF pattern

An open reading frame (ORF) starts with `ATG`, continues in whole codons, and ends at the first in-frame stop codon. As a pattern:

```
ATG (?:[ACGT]{3})*? (?:TAA|TAG|TGA)
```

(written without the spaces). `(?:[ACGT]{3})*?` is "any number of codons, as few as possible". Because it moves 3 bases at a time, the stop codon must be in the same frame as the `ATG`, and the lazy `*?` stops at the first one.

```python
orf_pat = re.compile(r"ATG(?:[ACGT]{3})*?(?:TAA|TAG|TGA)")
dna = "CCATGAAACCCTGAGGATGTTTTAGCC"
for m in orf_pat.finditer(dna):
    print(m.start() + 1, m.end(), m.group(), len(m.group()) // 3, "codons")

long_orfs = [m for m in orf_pat.finditer(chrI) if len(m.group()) >= 300]
print(len(long_orfs), "ATG..stop stretches of 300+ bp on the + strand of chrI")
```

```
3 14 ATGAAACCCTGA 4 codons
17 25 ATGTTTTAG 3 codons
56 ATG..stop stretches of 300+ bp on the + strand of chrI
```

This is a nice exercise, but a poor gene finder:

- matches cannot overlap, so an ORF that starts inside another one is missed (you would need the lookahead trick, and then you get many nested ORFs)
- it only reads the forward strand (search `revcomp(chrI)` too)
- it knows nothing about introns, alternative start codons or the mitochondrial genetic code

Real ORF finding reads each of the six frames codon by codon (e.g. Biopython's `Seq.translate()`, EMBOSS `getorf`, NCBI ORFfinder). Regex is still great for **checking** sequences. Do all the annotated yeast ORFs start with ATG, have a whole number of codons, and end with a stop?

```python
orfs = read_fasta("S_cerevisiae.ORFs.fasta.gz")
cds_pat = re.compile(r"ATG(?:[ACGT]{3})*(?:TAA|TAG|TGA)")
odd = []
for name, s in orfs.items():
    if not cds_pat.fullmatch(s):
        odd.append(name)
print(len(orfs) - len(odd), "of", len(orfs), "ORFs look like complete CDSs")
print(odd)
```

```
6704 of 6713 ORFs look like complete CDSs
['YAR061W', 'YFL056C', 'YIL175W', 'Q0010', 'Q0032', 'Q0075', 'Q0092', 'Q0144', 'Q0182']
```

The `Q0...` ORFs are mitochondrial genes, which use a different genetic code (for example `ATA` can be a start codon and `TGA` codes for tryptophan).

12. Python regex vs `grep -E` and `sed`

`grep -E` (extended regular expressions) and `sed` use nearly the same syntax. The main differences: GNU `grep` does not know `\d` (use `[0-9]`), lazy `*?` and lookahead need `grep -P`, and `sed` uses `\1` in replacements just like `re.sub`.

Task	bash	Python
lines that match	`grep -E 'TAA|TAG' file`	`if re.search(r"TAA|TAG", line):`
count matching lines	`grep -c '^>' file`	`if line.startswith(">"): n += 1`
whole word	`grep -w Chr1`	`re.search(r"\bChr1\b", line)`
ignore case	`grep -i gaattc`	`re.search(r"gaattc", line, flags=re.I)`
only the matching part	`grep -o -E '[0-9]+'`	`re.findall(r"\d+", line)`
replace	`sed 's/Chr/chr/g'`	`re.sub(r"Chr", "chr", line)`
replace with groups	`sed -E 's/(.+)_(R[12])/\2_\1/'`	`re.sub(r"(.+)_(R[12])", r"\2_\1", s)`

Use the shell for a quick look at a file; use Python when you need to keep the pieces (in variables, dictionaries or tables) and compute with them.

13. When not to use a regex

Regular expressions are great for small, messy or loosely structured pieces of text. For **well-defined file formats**, a proper parser is simpler and safer:

- tab or comma separated tables: `line.split("\t")` or the `csv` module (Python II); `pandas.read_csv` (Pandas)
- FASTA, FASTQ, GenBank: `Bio.SeqIO` from Biopython (Python IV)
- GFF/GTF, VCF, SAM/BAM, JSON, XML: use a library written for the format

A good pattern is: let the parser split the file into fields, then use a small regex on one field (a description line, a GFF attribute column, a sample name).

Common mistakes

- **Forgetting the `r`**: `"\bGAATTC\b"` silently contains backspace characters. Always write `r"..."`.
- **Using `.group()` on `None`**: test `if m:` before using the match.
- **`re.match` when you meant `re.search`**: `re.match` only looks at the start of the string. To validate a whole string use `re.fullmatch`.
- **Greedy `.*`** runs to the *last* possible match. Use `[^;]*` or `.*?`.
- **Forgetting to escape `.`, `|`, `(`, `+`, `?`**: `YP_488307.1` also matches `YP_488307x1`; write `YP_488307\.1` or use `re.escape()`.
- **Unprotected `|`**: `^chrI|II$` means `^chrI` or `II$`. Write `^chr(?:I|II)$`.
- **`findall` with groups** returns only the groups. Use `(?:...)` if you want the whole match.
- **Flags in the wrong position**: write `flags=re.I`.
- **Off-by-one coordinates**: `m.start()` is 0-based; add 1 for GFF/VCF style.
- **Overlaps**: `findall`, `finditer` and `str.count` never report overlapping matches; use `(?=(...))`.
- **Case**: `GAATTC` does not match soft-masked `gaattc`; use `re.I` or `.upper()` the sequence first.

Quick reference

Syntax	Meaning	Example
`abc`	the literal text abc	`GAATTC`
`.`	any character except newline	`T.A`
`[ACGT]`	one character from the set	`GT[CT][AG]AC`
`[^ACGT]`	one character NOT in the set	`[^;]+`
`[a-z]`, `[0-9]`	ranges	`[A-Za-z]`
`\d`, `\D`	digit, not a digit	`\d+`
`\w`, `\W`	word character `[A-Za-z0-9_]`, not one	`\w+`
`\s`, `\S`	whitespace, not whitespace	`\S+`
`\.`	a literal dot (escape any special character)	`\d+\.\d`
`^`, `$`	start, end of string	`^>`
`\b`	word boundary	`\bChr1\b`
`*`, `+`, `?`	0 or more, 1 or more, 0 or 1	`reads?`
`{n}`, `{n,m}`, `{n,}`	exactly n, n to m, n or more	`A{4,}`
`*?`, `+?`	lazy: as few as possible	`ID=(.*?);`
`A|B`	A or B	`TAA|TAG|TGA`
`(...)`	group and capture	`(\d+)-(\d+)`
`(?P<name>...)`	named group	`(?P<chrom>\w+)`
`(?:...)`	group without capturing	`(?:CA){3,}`
`(?=...)`	lookahead (does not consume)	`(?=(ATA))`

Function	Returns
`re.search(p, s)`	first match anywhere, or `None`
`re.match(p, s)`	match at the start of `s`, or `None`
`re.fullmatch(p, s)`	match of the whole `s`, or `None`
`re.findall(p, s)`	list of matched strings (or of groups)
`re.finditer(p, s)`	match objects, one per match (with positions)
`re.sub(p, repl, s)`	new string with matches replaced
`re.split(p, s)`	list of the pieces between matches
`re.compile(p, flags)`	a pattern object with the same methods
`m.group(n)`, `m.groups()`, `m.groupdict()`	the captured text
`m.start()`, `m.end()`, `m.span()`	positions (0-based, end excluded)

Exercises

Solutions are in the worked examples. Try them yourself first!

1. **Validate sequences.** Write a function `is_dna(seq)` that returns `True` only if `seq` contains nothing but A, C, G, T (upper or lower case), and `is_protein(seq)` for the 20 standard amino acid letters. Test them on `"ACGTN"`, `"acgt"`, `"MKRISTT"` and `"MKRIS*"`.
2. **Count EcoRI sites in every chromosome.** Using `read_fasta()` on `S_cerevisiae.fasta.gz`, print each chromosome's name, length, number of EcoRI sites and sites per 10 kb. Hint: `re.findall` and `len()`.
3. **Sample names.** From this list, print the sample name and read number for the paired FASTQ files only, and ignore the others: `["sampleA_R1.fastq.gz", "sampleA_R2.fastq.gz", "ctrl-2_R1.fq.gz", "sampleA_R1.fastq.gz.md5", "notes.txt"]`. Hint: `re.fullmatch` and two groups.
4. **Accessions.** Find all RefSeq assembly accessions (`GCF_` + 9 digits + `.` + version) and all SRA run accessions (`SRR`, `ERR` or `DRR` + digits) in: `"Reads SRR1234567 and ERR998877 were mapped to GCF_000146045.2 (R64); see also GCA_000146045.2 and SRR12."`
5. **Degenerate motif, both strands.** The SCB cell-cycle element bound by SBF (Swi4/Swi6) is `CRCGAAA`. How many sites are on chromosome III, on each strand? Use `find_motif()`. Is the motif palindromic?
6. **Overlapping motifs.** Count `TATA` in `"TATATATA"` with `str.count`, with `re.findall` and with a lookahead. Then count overlapping occurrences of the TATA-box-like motif `TATAWAWR` on chromosome I.

7. **GFF3 coordinates.** Using `re.sub` and groups, convert SGD-style locations such as `"Chr IV from 1802-2953"` into `"chrIV:1802-2953"`, and turn `"chrIV:1802-2953"` back into a tuple `("chrIV", 1802, 2953)` of a string and two integers.
8. **Protein motif.** The N-glycosylation motif is `N`, then anything but `P`, then `S` or `T`, then anything but `P` (PROSITE `N-{P}-[ST]-{P}`). Write it as a regex and count, with overlaps, how many E. coli proteins in `E_coli_K12.pep.gz` contain at least one site.

Tools and further reading

- regex101.com - paste a pattern and some text, and it explains every part and shows the matches. Choose the **Python** flavor on the left.
- The Python Regular Expression HOWTO and the `re` module documentation.
- The UNIX III data processing lab for `grep -E` and `sed`.

Next

- Regular Expression Worked Examples: parsing real FASTA headers and GFF3 files, file names, accession numbers, messy tables, and the exercise solutions.
- Python IV: Packages: Biopython for FASTA/GenBank parsing.
- Pandas: `df["col"].str.contains()`, `.str.extract()` and `.str.replace()` apply regular expressions to a whole column at once.
- Plotting: plot what you extracted, e.g. the distribution of motif counts per chromosome.