Regular Expressions in Python — **Detailed Student Notes**

Overview — what these notes cover

These notes teach you how to read, write, and use **regular expressions (regex)** in Python. You'll learn:

- What regex is and why it's useful
- Basic regex syntax and special characters
- Character classes, shorthand classes, and negation
- Quantifiers (greedy vs lazy)
- Grouping, capturing, backreferences, and named groups
- Searching vs matching vs full matching
- Finding multiple matches (`findall`, `finditer`)
- Substitution using `re.sub` and using functions as replacers
- Compiling patterns for reuse and performance
- Useful flags (case-insensitive, verbose, multiline)
- Practical examples and exercises (with solutions)
- A one-page cheat-sheet at the end for quick revision

Python module used throughout: `re` (standard library). Use **raw strings** (prefix `r''`) for patterns to avoid Python string escapes interfering with regex backslashes.

Why learn regex?

Regex is a compact language for describing text patterns. Real examples where regex helps:

- Extracting phone numbers, emails, or hashtags from text
- Validating simple input (dates, simple email forms)
- Cleaning messy text (collapsing whitespace, removing duplicates)
- Transforming URLs or log lines into structured data

Regex can replace many lines of custom parsing code with short, reliable patterns — but it has limits (e.g., nested HTML/XML is better handled with proper parsers).

Setup / Tools

- Python 3.x (3.8+ recommended but any 3.x works).
- `re` module (built-in — no pip).
- Code editor or REPL (VS Code, PyCharm, IDLE, or an online REPL).
- Optional: regex tester sites (regex101), but practice in Python to see exact behavior.

Quick start — first examples

```
import re

text = 'The cat sat on the carpet.'

print(re.search(r'cat', text))   # returns a match object if found
print(re.search(r'dog', text))   # returns None if not found

# re.match checks at the start only:
print(bool(re.match(r'The', text)))  # True
print(bool(re.match(r'cat', text)))  # False
```

Notes: - `re.search` scans the whole string and returns the first match. - `re.match` only matches at the beginning of the string. - Use `r'pattern'` (raw string) to avoid double escaping.

Core regex symbols & meanings (short summary)

- `.` — any character except newline (by default).
- ^ — start of string (or start of line with `re.M`).
- $ — end of string (or end of line with `re.M`).
- [ ] — character class (match any single char inside).
- `\d` — digit (0–9). `\D` — non-digit.
- `\w` — word character (letters, digits, underscore). `\W` — non-word.
- `\s` — whitespace (space, tab, newline). `\S` — non-whitespace.
- * — 0 or more of previous item (greedy).
- + — 1 or more (greedy).
- ? — 0 or 1 (also makes quantifiers lazy when appended to them).
- `{m}`, `{m,}`, `{m,n}` — exact/at-least/bounded counts.
- ( ) — group and capture.

- | — OR (alternation).
- \ — escape special characters or introduce sequences (\d, \b, \1, ...).
- \b — word boundary.

Special characters and escaping

Some characters have special meanings. To match them literally, escape with \.

Examples:

```
re.search(r'\.', 'Hello.')   # matches a literal dot
re.search(r'\$', '$100')     # matches literal dollar sign
```

Tip: always write regex patterns as raw strings in Python: r'\.' rather than '\\.'.

Character classes and shorthand classes

Character classes — []

- [abc] — matches 'a' or 'b' or 'c'.
- [a-z] — any lowercase letter.
- [A-Za-z0-9] — letters and digits.
- [^abc] — negation — match any character *except* a, b, or c.

Examples:

```
re.findall(r'[aeiou]', 'Regular expressions are fun')   # finds vowels
re.findall(r'[0-9A-Fa-f]', '0xAf3')                     # hex digits
```

Shorthand classes

- \d — digit ([0-9])
- \w — word character ([A-Za-z0-9_])
- \s — whitespace (spaces, tabs, newlines)
- Uppercase versions (\D, \W, \S) are negations.

Important: \w includes underscore _ and only ASCII letters by default; use re.UNICODE behavior for Unicode word chars (Python 3 default is Unicode-aware, but semantics differ for punctuation).

Quantifiers — how many times?

- * — zero or more of previous atom

- `+` — one or more
- `?` — zero or one
- `{m}`, `{m,}`, `{m,n}` — custom counts

By default quantifiers are **greedy**: they match as much as possible. Use `?` after quantifier to make it **lazy** (non-greedy):

- Greedy: `.*` — as much as possible
- Lazy: `.*?` — as little as possible while still allowing the overall match

Example demonstrating greedy vs lazy:

```
s = '<p>first</p><p>second</p>'

# greedy: matches from first <p> to last </p>
re.search(r'<p>.*</p>', s).group()
# returns '<p>first</p><p>second</p>'

# lazy: matches the smallest <p>...</p>
re.search(r'<p>.*?</p>', s).group()
# returns '<p>first</p>'
```

Analogy: greedy = vacuum cleaner; lazy = picky eater.

Grouping, capturing, and backreferences

Grouping with `()`

Parentheses group parts of a pattern. If you capture a group (normal parentheses), Python stores the captured text and you can retrieve it:

```
m = re.search(r'([\w.+-]+)@([\w.-]+)', 'sam@example.com')
m.groups()           # ('sam', 'example.com')
m.group(1)           # 'sam'
m.group(2)           # 'example.com'
```

Backreferences — `\1`, `\2`, ...

Backreferences let you match the exact text captured earlier:

```
re.findall(r'\b(\w+)\s+\1\b', 'This is is a test. That that', flags=re.I)
# ['is', 'that'] – finds duplicated consecutive words
```

Named groups — `(?P<name>...)`

Use names for clarity and readability:

```
m = re.search(r'(?P<user>[\w.+-]+)@(?P<domain>[\w.-]+)', 'sam@example.com')
m.group('user')   # 'sam'
m.group('domain') # 'example.com'
```

Use backrefs with names in replacement strings: e.g., `r'\g<name>'` in `re.sub`.

`re` functions — quick reference

- `re.search(pattern, string)` — find first match anywhere (returns match object or None).
- `re.match(pattern, string)` — match at start only.
- `re.fullmatch(pattern, string)` — entire string must match pattern.
- `re.findall(pattern, string)` — list of all non-overlapping matches (strings or tuples).
- `re.finditer(pattern, string)` — iterator over match objects (useful when you need positions or group data).
- `re.sub(pattern, repl, string)` — replace matches with `repl` (string or function).
- `re.compile(pattern, flags=0)` — compile regex into a pattern object for reuse (call `.search`, `.sub`, etc. on it).

Examples:

```
re.findall(r'\d+', 'Call me on 080-555-1234 or 081 999 0000')
# ['080', '555', '1234', '081', '999', '0000']

for m in re.finditer(r'\d+', 'abc 123 def 456'):
    print(m.group(), m.span())
```

Substitution: `re.sub`

`re.sub` replaces occurrences of a pattern with a replacement string or function.

Examples:

```
# collapse repeated whitespace to single space
s = 'This  text   has    irregular   spacing.'
clean = re.sub(r'\s+', ' ', s).strip()
print(clean)  # 'This text has irregular spacing.'

# remove duplicated adjacent words:
text = 'this is is a test test'
print(re.sub(r'\b(\w+)\s+\1\b', r'\1', text, flags=re.I))
# 'this is a test'
```

```python
# using a function as replacement (mask digits)
def mask(match):
    return 'X' * len(match.group(0))

print(re.sub(r'\d', mask, 'Call 080-555-1234'))
# digits replaced with Xs
```

When using group backreferences in replacement strings, use `r'\1'` (raw string) or `\\1` in non-raw strings. For named groups in replacement use `r'\g<name>'`.

Compiling patterns: `re.compile`

If you reuse a pattern many times, compile it once:

```python
phone_re = re.compile(r'(\+?\d{1,3})?[-.\s]?(\d{3})[-.\s]?(\d{3})[-
.\s]?(\d{4})')
m = phone_re.search('Call +234-803-555-1234')
m.groups()  # ('+234', '803', '555', '1234')
```

You can pass flags during compile: `re.compile(r'pattern', re.I|re.X)`.

Useful flags

- `re.I` or `re.IGNORECASE` — case-insensitive matching.
- `re.M` or `re.MULTILINE` — ^ and $ match at start/end of each line.
- `re.S` or `re.DOTALL` — `.` matches newline as well.
- `re.X` or `re.VERBOSE` — ignore whitespace and allow comments inside patterns (very useful for readability).

Example with verbose flag:

```python
pattern = re.compile(r'''
    (?:\+?234)?      # optional country code
    [-.\s]?          # optional separator
    (\d{3})          # area
    [-.\s]?
    (\d{3})
    [-.\s]?
    (\d{4})
''', re.X)
```

`re.X` allows you to write readable, commented regexes across multiple lines.

Practical examples (explained)

1) Find mentions `@username` and hashtags `#tag`

```
s = "Hey @Sam! See ##party #FreeFood"
mentions = re.findall(r'@[\w]+', s)
hashtags = re.findall(r'#\w+', s)
# mentions => ['@Sam']
# hashtags => ['#party', '#FreeFood'] (note: '##party' yields '#party'
depending on pattern)
```

2) Find phone numbers and normalize to `+234XXXXXXXXXX`

```
def norm_phone(m):
    all_digits = re.sub(r'\D', '', m.group(0))  # strip non-digits
    if all_digits.startswith('234') and len(all_digits) == 13:
        return '+' + all_digits
    if len(all_digits) == 11 and all_digits.startswith('0'):
        return '+234' + all_digits[1:]
    return m.group(0)

pattern = r'(?:\+?234[-.\s]?\d{3}[-.\s]?\d{3}[-.\s]?\d{4})|(?:0\d{2}[-
.\s]?\d{3}[-.\s]?\d{4})'
normalized = re.sub(pattern, norm_phone, text)
```

3) Remove duplicate adjacent words and collapse overlong letter runs

```
# remove duplicate words (case-insensitive)
s = re.sub(r'\b(\w+)\s+\1\b', r'\1', s, flags=re.I)

# collapse triple-or-more identical letters into a single letter
s = re.sub(r'(\w)\1{2,}', r'\1', s)   # 'coool' -> 'col' or 'coool' -> 'col'?
see note below
```

Exercises (practice)

Input string for exercises

```
Hey @Sam! Please RSVP by 05/09/2025. Also: call +234 803 555 1234 or 080-555-
1234.
Visit https://example.com/party?invite=ABC123. ##party #FreeFood
P.S. I accidentally typed the the in the message, and I've got coool cake.
See you there!
```

Exercise A — Easy

Find all mentions and hashtags.

```
mentions = re.findall(r'@[\w]+', s)
hashtags = re.findall(r'#\w+', s)
```

Exercise B — Medium

Find all phone numbers and normalize them to `+234XXXXXXXXXX`.

```
normalized = re.sub(pattern, norm_phone, s)
re.findall(r'\+234\d{10}', normalized)
```

Exercise C — Medium+

Remove duplicate adjacent words and collapse overlong letters (3+ repeats) to a single instance.

```
s2 = re.sub(r'\b(\w+)\s+\1\b', r'\1', s, flags=re.I)
s2 = re.sub(r'(\w)\1{2,}', r'\1', s2)
```

Exercise D — Challenging

Capture the invite code from URL `invite=ABC123` and validate it is 3 letters followed by 3 digits; then replace with `REDACTED`.

```
m = re.search(r'invite=([A-Z]{3}\d{3})', s)
invite_code = m.group(1)
redacted = re.sub(r'(invite=)([A-Z]{3}\d{3})', r'\1REDACTED', s)
```

Common pitfalls & best practices

1. Don't use regex for nested grammars.
2. Make patterns readable with `re.X`.
3. Test incrementally.
4. Use named groups for clarity.
5. Be careful with greedy quantifiers.
6. Escape special characters.
7. Normalize before matching.

Advanced tips

- Use `re.finditer()` for match positions or details.
- Use function replacers in `re.sub()` for logic-based replacements.
- Combine flags: `re.compile(pattern, re.I | re.X)`.
- Use lookahead/lookbehind for zero-width assertions: `(?=...)`, `(?!...)`, `(?<=...)`, `(?<!...)`.
- Use `re.fullmatch` for full-string validation.

One-page cheat-sheet

- **Raw string**: `r'pattern'`
- **Functions**: `re.search`, `re.match`, `re.fullmatch`, `re.findall`, `re.finditer`, `re.sub`, `re.compile`
- **Special chars**: `. ^ $ \ | ( ) [ ] { } * + ?`
- **Shorthand**: `\d` digit, `\w` word char, `\s` whitespace; uppercase = negation
- **Quantifiers**: `* + ? {m} {m,} {m,n}` — append `?` for lazy: `*?`
- **Groups**: `()` capture, `(?:)` non-capturing, `(?P<name>)` named
- **Backrefs**: `\1`, `\2` or `\g<name>`
- **Flags**: `re.I`, `re.M`, `re.S`, `re.X`
- **Substitute**: `re.sub(pattern, repl, s)` (repl can be function)

Additional practice problems

1. Extract all domain names from URLs.
2. Validate dates in `DD/MM/YYYY` format.
3. Extract quoted text from a paragraph (both single and double quotes).
4. Convert dates from DD/MM/YYYY to YYYY-MM-DD using `re.sub` and a replacer function.
5. Write a regex to find words longer than 12 letters.

Solutions to homework (short)

1. Extract all domain names from URLs.
2. Domain extraction: `r'https?://(?:www\.)?([^/]+)'` then `.group(1)`.
3. Date basic: `r'\b(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/(\d{4})\b'`
4. Quoted text: `r'(["\'])(?:(?=(\\?))\2.)*?\1'` (a generic approach — nuances exist) — simpler: `r'"([^"]*)"|\'([^\']*)\''`
5. Reformat date with function:

```
def repl(m):
    d, mth, y = m.group(1), m.group(2), m.group(3)

    return f'{y}-{mth}-{d}'

re.sub(r'\b(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/(\d{4})\b', repl, text)
```

Final encouragement

Regex is a practical, high-leverage skill for any developer who works with text data. Start simple, test often, and prefer readability (verbose mode) for complex patterns. When a task gets hairy (nested tags, deep structure), reach for a parser.