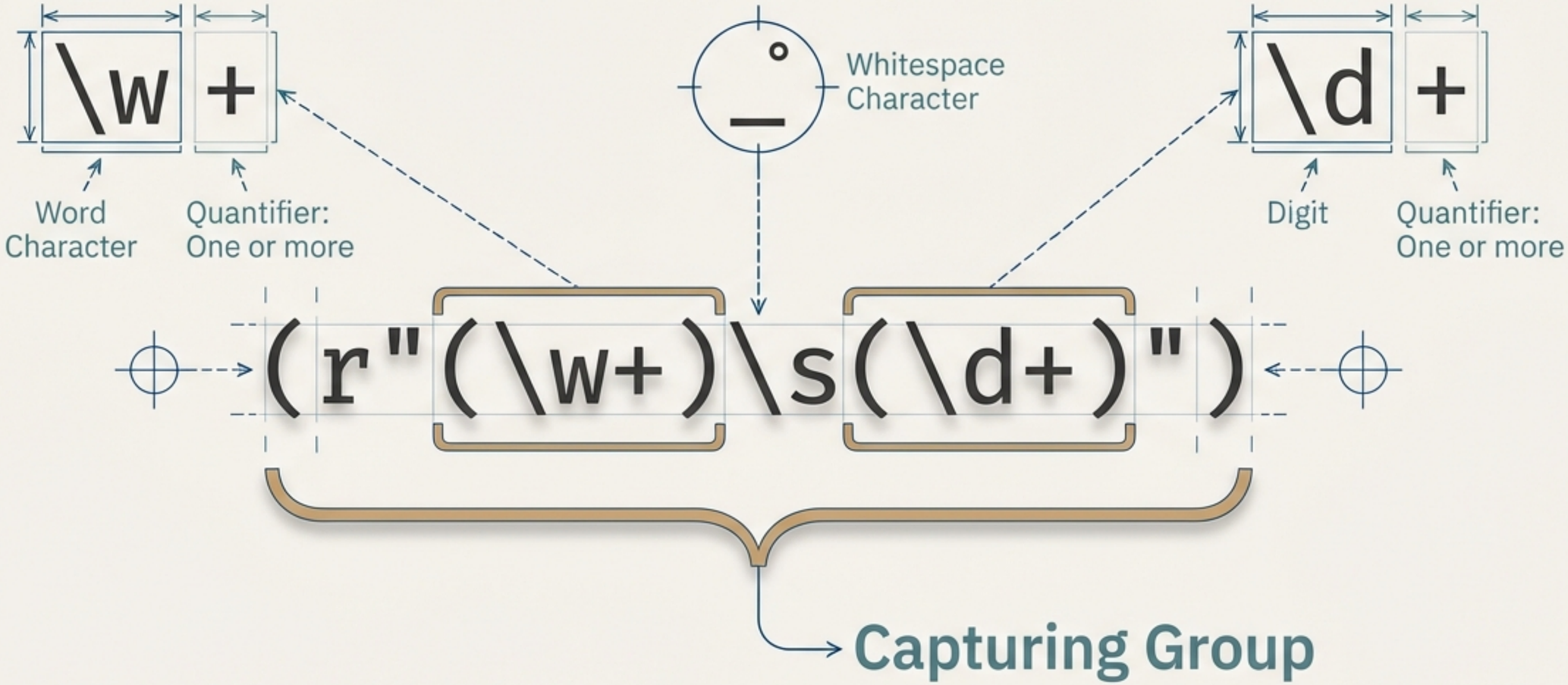
MASTERING TEXT: A BLUEPRINT FOR PYTHON REGULAR EXPRESSIONS

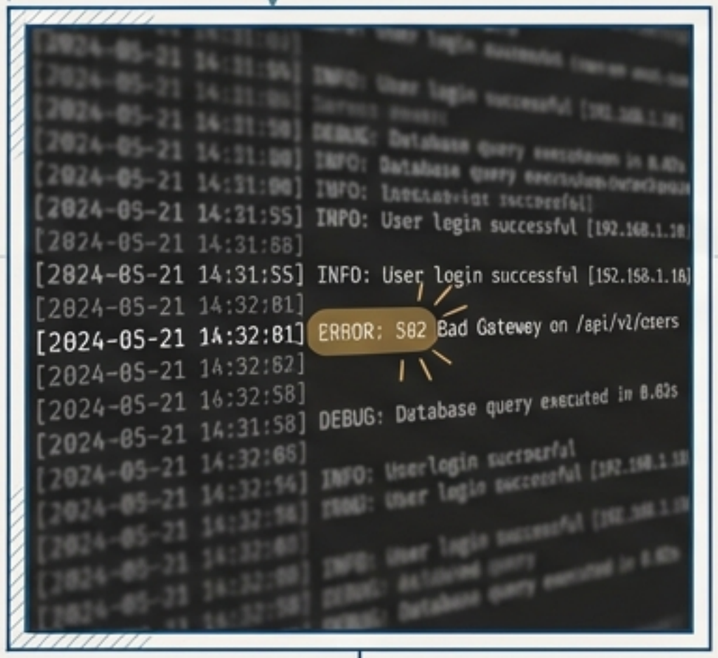
A practical guide to finding, validating, and transforming text data with precision.



Every developer works with text. But most of it is unstructured.

We constantly face raw text from log files, user inputs, web pages, and APIs. How do we find the signal in the noise and shape it into valuable, structured data?

Log Files



Web Content

```
<html>
 <head>
   <title>Example Page</title>
 </head>
 <body>
   <div id="main-content" class="containe
     <p>Welcome to our website. For
     inquiries, please <a
   href="mailto:contact-us@example.com">
contact us</a> or call <span
      class="phone">+1-555-0123</span>.
   </p>
   </div>
   <footer>...</footer>
 </body>
</html>
```

User Input

Registration Date
21-05-2024
May 22, 2024
23/05/24
2024-05-24

Target Format (ISO 8601)

Regular Expressions are the **standard** for tackling these challenges.

The Solution: A Universal Language for Text Patterns

A **Regular Expression** (or Regex) is a special sequence of characters that defines a search pattern. It's a powerful tool for matching, searching, and manipulating strings.

In Python, we access this power through the built-in `re` module.

```
# The only import you need to get started
import re
```

The First Cut: Finding Your First Match

`re.search()` - Finds a match anywhere

Scans the entire string and returns the *first* location where the pattern produces a match.

```
text = "My number is 9876543210"
match = re.search(r"\d+", text)
print(match.group())
```

Result:

```
'9876543210'
```

`re.match()` - Only matches at the beginning

Checks for a match only at the *beginning* of the string.

```
# This will find a match
re.match(r"Hello", "Hello Python")

# This will not
re.match(r"Hello", "Python Hello")
```

✓ “Finds a match in ‘Hello Python’”

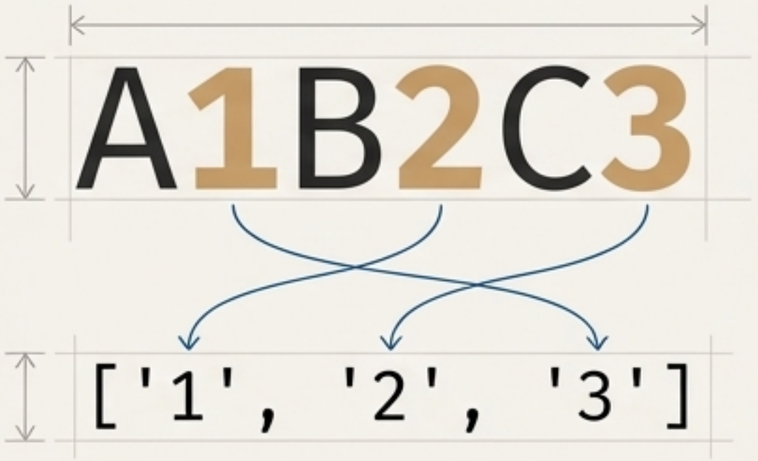
✗ “Returns None for ‘Python Hello’”

Beyond the First Match: Finding All and Replacing

re.findall() – Extracting all matches

Finds all non-overlapping matches of the pattern in the string and returns them as a list.

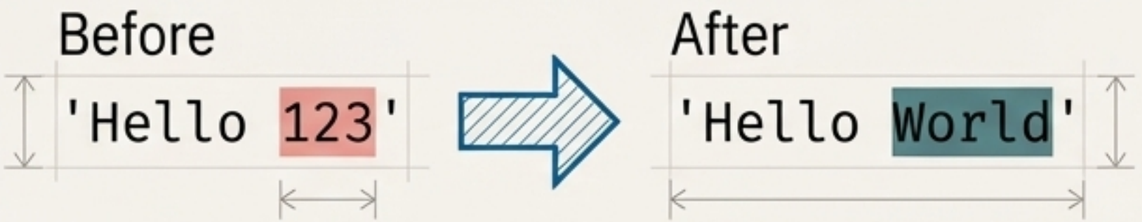
```
re.findall(r"\d", "A1B2C3")
```



re.sub() - Finding and replacing text

Replaces all occurrences of the pattern in the string with a specified replacement.

```
text = "Hello 123"
re.sub(r"\d+", "World", text)
```



The Language of Patterns: Special Characters

These single characters are shorthands for common patterns. Think of them as the alphabet of regex.

`\d` : Any digit (0-9). `\D` : Not a digit.	`.` : Any character (except newline).	`[...]` : Custom character set.
`\w` : Any alphanumeric character (a-z, A-Z, 0-9, _) `\W` : Not alphanumeric.	`^` : Start of the string.	`(...)` : Capturing group.
`\s` : Any whitespace character. `\S` : Not whitespace.	`$` : End of the string.	`|` : Alternation (OR).

Controlling Repetition: Quantifiers

Specify exactly how many times a character, group, or class should be matched.

Pattern	Meaning
`+`	One or more
`*`	Zero or more
`?`	Zero or one
`{n}`	Exactly *n* times
`{n,}`	At least *n* times
`{n,m}`	Between *n* and *m* times

Concept-Example-Result

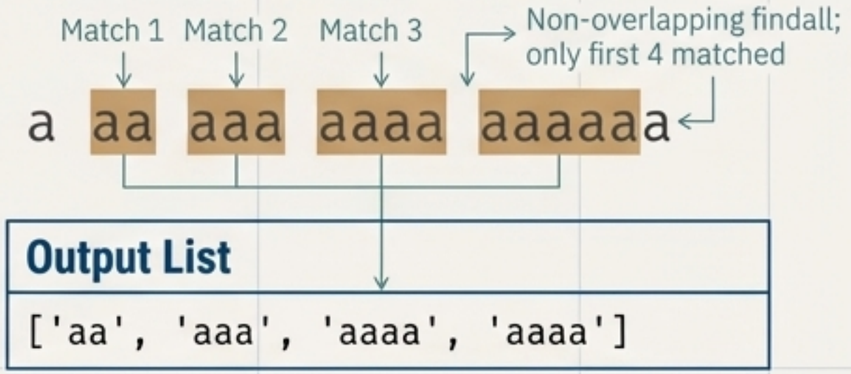
Concept

To match the letter ‘a’ appearing between 2 and 4 times.

Example

```
re.findall(r"a{2,4}", "a aa aaa aaaa aaaaa")
```

Result



Defining Your Own Rules: Character Classes [ ]

Square brackets [ ] let you create a custom set of characters to match. Any single character from within the brackets will match.

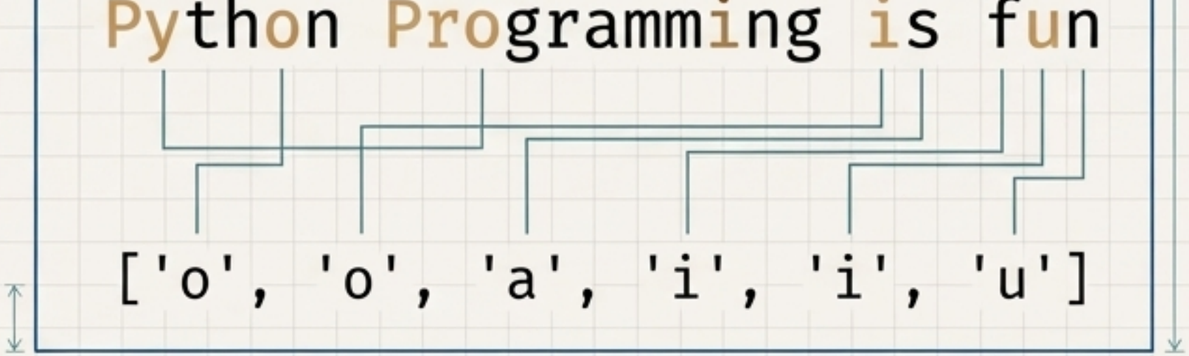
Common Ranges

- `[a-z]`: Matches any lowercase letter.
- `[A-Z]`: Matches any uppercase letter.
- `[0-9]`: Matches any digit.
- `[aeiou]`: Matches any vowel.

Example

Task: Find all vowels in a string.

```
re.findall(r"[aeiou]", "Python
 Programming is fun")
```



Capturing and Extracting: Grouping with `( )`

Parentheses `( )` serve two purposes:

1. **Group Patterns:** Treat multiple characters as a single unit (e.g., `(abc)+`).
2. **Capture Substrings:** Extract the specific part of the string that matched the pattern inside the parentheses.

Example

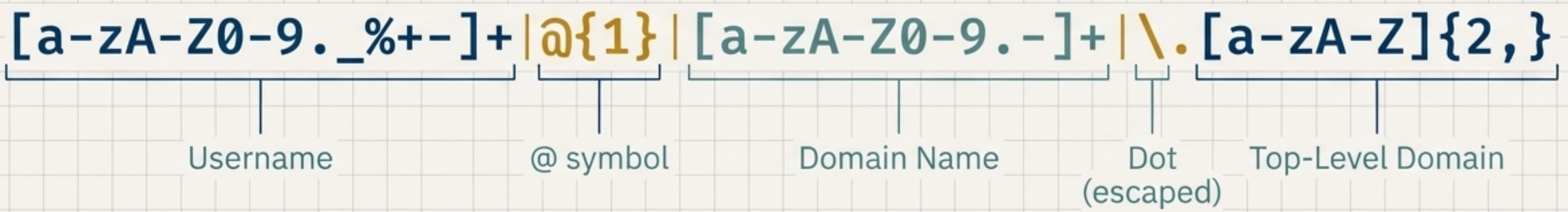
Task: Extract only the titles (Mr, Ms, Dr) from a list of names.

```
pattern = r"(Mr|Ms|Dr)\.[A-Z][a-z]+"
text = "Dr. John met Mr. David and Ms. Smith"
re.findall(pattern, text)
```


```
Final Result: ['Dr', 'Mr', 'Ms']
```

Key Insight: Notice how `findall()` returns only the content from the captured group `(Mr|Ms|Dr)`, not the entire matched string.

Practical Application: Extracting Email Addresses



Code in Action IBM Plex Sans (Charcoal Grey)

```
text = "Contact support at help@example.co.uk or sales@company.com for details."
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"
emails = re.findall(pattern, text)
print(emails)
```

```
Result:
['help@example.co.uk', 'sales@company.com']
```

Advanced Craftsmanship: Validating User Input

Use anchored patterns (`^` and `$`) to ensure the *entire* string conforms to a specific format.

Username Validation

Must be 3-15 characters long and contain only letters, numbers, or underscore.

```
r"^[a-zA-Z0-9_]{3,15}$"
```

- `^` : Starts at the beginning of the string.
- `[...]` : Allowed characters (alphanumeric & underscore).
- `{3,15}` : Quantifier, must be 3 to 15 characters.
- `$` : Ends at the end of the string.

Password Validation (Example)

At least 6 characters, one uppercase letter, and one digit.

```
r"^(?=.*[A-Z])(?=.*\d).{6,}$"
```

- `^...$` : Anchors the whole pattern.
- `(?=.*[A-Z])` : A “positive lookahead” that asserts an uppercase letter exists, without consuming characters.
- `(?=.*\d)` : A lookahead asserting a digit exists.
- `.{6,}` : Ensures the total length is at least 6.

Optimising Your Tools: Performance and Flags

Improve Performance with re.compile()

If you use the same pattern multiple times, compiling it first creates a reusable pattern object that is more efficient.

```
# Compile once, for efficiency
pattern = re.compile(r"\d{4}")

# Use the compiled object many times
pattern.findall("Year: 2024 Month: 12")
pattern.findall("Another year is 2025")
```

Creates reusable object

Efficient multiple uses

Modify Behaviour with Flags

Flags are optional arguments that change how a pattern is interpreted.

Without Flag (Case-Sensitive)

```
# This would normally fail
re.findall(r"python", "PYTHON")
```

```
[]
```

With `re.I` Flag (Case-Insensitive)

```
# The re.I flag makes it match
re.findall(r"python", "PYTHON", re.I)
```

```
['PYTHON']
```

Other useful flags: `re.M` for multi-line matching, `re.S` for making `.` match newlines.

The Regex Craftsman's Cheat Sheet

Core Functions

`re.search(p, s)`	Finds first match anywhere.
`re.match(p, s)`	Finds first match at start.
`re.findall(p, s)`	Returns all matches as a list.
`re.sub(p, r, s)`	Replaces pattern `p` with `r` in string `s`.

Essential Syntax

`[...]`	Character set (e.g., `[abc]`).
`(...)`	Capturing group.
`|`	OR operator (e.g., `cat|dog`).
`\`	Escape special character (e.g., `\.`).

Common Metacharacters

`\d / \D`	Digit / Non-digit.
`\w / \W`	Word char / Non-word char.
`\s / \S`	Whitespace / Non-whitespace.
`^ / $`	Start / End of string.

Common Quantifiers

`+`	One or more.
`*`	Zero or more.
`?`	Zero or one.
`{n,m}`	Between n and m.

From Raw Text to Structured Insight

Name	Email	ID
Alice Smith	alice@example.com	1001
Bob Jones	bob.jones@mail.net	1002
Charlie Brown	charlie.brown@web.org	1003

Regular expressions are more than just syntax; they are a fundamental tool for any developer working with text. Mastering them allows you to find, validate, and transform data with precision and efficiency.