In [1]:
```
from dsc80_utils import *
```

Lecture 11 – Regular Expressions

DSC 80, Fall 2025

Agenda 📆

- Most of today's lecture will be about **regular expressions**. Good resources:
  - regex101.com, a helpful site to have open while writing regular expressions.
  - Python `re` library documentation and how-to.
    - The "how-to" is great, read it!
  - regex "cheat sheet" (taken from here).
  - These are all on the **resources tab of the course website** as well.

Motivation

In [2]:
```
contact = '''
Thank you for buying our expensive product!

If you have a complaint, please send it to complaints@compuserve.com or call

If you are happy with your purchase, please call us at (800) 123-4567; we'd

Due to high demand, please allow one-hundred (100) business days for a respo
'''
```

Who called? 📞

- **Goal**: Extract all phone numbers from a piece of text, assuming they are of the form `'(###) ###-####'`.

In [3]:
```
print(contact)
```

```
Thank you for buying our expensive product!

If you have a complaint, please send it to complaints@compuserve.com or call
(800) 867-5309.

If you are happy with your purchase, please call us at (800) 123-4567; we'd
love to hear from you!

Due to high demand, please allow one-hundred (100) business days for a respo
nse.
```

- We can do this using the same string methods we've come to know and love.
- Strategy:
  - Split by spaces.
  - Check if there are any consecutive "words" where:
    - the first "word" looks like an area code, like `'(678)'`.
    - the second "word" looks like the last 7 digits of a phone number, like `'999-8212'`.

Let's first write a function that takes in a string and returns whether it looks like an area code.

In [4]:
```
def is_possibly_area_code(s):
    '''Does `s` look like (678)?'''
    return (len(s) == 5 and
            s.startswith('(') and
            s.endswith(')') and
            s[1:4].isnumeric())
```

In [5]:
```
is_possibly_area_code('(123)')
```

Out[5]:
```
True
```

In [6]:
```
is_possibly_area_code('(99)')
```

Out[6]:
```
False
```

Let's also write a function that takes in a string and returns whether it looks like the last 7 digits of a phone number.

In [7]:
```
def is_last_7_phone_number(s):
    '''Does `s` look like 999-8212?'''
    return len(s) == 8 and s[0:3].isnumeric() and s[3] == '-' and s[4:].isnu
```

In [8]:
```
is_last_7_phone_number('999-8212')
```

Out[8]:
```
True
```

In [9]:
```
is_last_7_phone_number('534 1100')
```

Out[9]:
```
False
```

Finally, let's split the entire text by spaces, and check whether there are any instances where `pieces[i]` looks like an area code and `pieces[i+1]` looks like the last 7 digits of a phone number.

```
# unctuation from the end of each string.
pieces = [s.rstrip('.,?;"\'') for s in contact.split()]
```

```
for i in range(len(pieces) - 1):
    if is_possibly_area_code(pieces[i]):
        if is_last_7_phone_number(pieces[i+1]):
            print(pieces[i], pieces[i+1])
```

```
(800) 867-5309
(800) 123-4567
```

Is there a better way?

- This was an example of **pattern matching**.
- It can be done with string methods, but there is often a better approach: **regular expressions**.

In [11]:
```
print(contact)
```

```
Thank you for buying our expensive product!

If you have a complaint, please send it to complaints@compuserve.com or call
(800) 867-5309.

If you are happy with your purchase, please call us at (800) 123-4567; we'd
love to hear from you!

Due to high demand, please allow one-hundred (100) business days for a respo
nse.
```

In [12]:
```
import re
re.findall(r'\(\d{3}\) \d{3}-\d{4}', contact)
```

Out[12]:
```
['(800) 867-5309', '(800) 123-4567']
```

Basic regular expressions

Regular expressions

- A regular expression, or **regex** for short, is a sequence of characters used to **match patterns in strings**.
  - For example, `\(\d{3}\) \d{3}-\d{4}` describes a **pattern** that matches US phone numbers of the form `'(XXX) XXX-XXXX'`.
  - Think of regex as a "mini-language" (formally: they are a grammar for describing a language).

- **Pros**: They are very powerful and are widely used (virtually every programming language has a module for working with them).

- **Cons**: They can be hard to read and have many different "dialects."

Writing regular expressions

- You will ultimately write most of your regular expressions in Python, using the `re` module. We will see how to do so shortly.
- However, a useful tool for designing regular expressions is regex101.com.
- We will use it heavily during lecture; you should have it open as we work through examples. **If you're trying to revisit this lecture in the future, you'll likely want to watch the podcast.**

Literals

- A literal is a character that has no special meaning.
- Letters, numbers, and some symbols are all literals.
- Some symbols, like `.`, `*`, `(`, and `)`, are special characters.
- ***Example***: The regex `hey` matches the string `'hey'`. The regex `he.` also matches the string `'hey'`.

Regex building blocks 🧰

The four main building blocks for all regexes are shown below (table source, inspiration).

operation	order of op.	example	matches ✅	does not match ❌
concatenation	3	`AABAAB`	`'AABAAB'`	every other string
or	4	`AA|BAAB`	`'AA'`, `'BAAB'`	every other string
closure (zero or more)	2	`AB*A`	`'AA'`, `'ABBBBBBA'`	`'AB'`, `'ABABA'`
parentheses	1	`A(A|B)AAB`	`'AAAAB'`, `'ABAAB'`	every other string
		`(AB)*A`	`'A'`, `'ABABABABA'`	`'AA'`, `'ABBA'`

Note that `|`, `(`, `)`, and `*` are **special characters**, not literals. They manipulate the characters around them.

Example (or, parentheses):

- What does `DSC 30|80` match?
- What does `DSC (30|80)` match?

Example (closure, parentheses):

- What does `blah*` match?
- What does `(blah)*` match?

> Question 🤔

Write a regular expression that matches `'billy'`, `'billlly'`, `'billllllly'`, etc.

- First, think about how to match strings with any even number of `'l'`s, including zero `'l'`s (i.e. `'biy'`).
- Then, think about how to match only strings with a **positive even** number of `'l'`s.

In [ ]:

> Question 🤔

Write a regular expression that matches `'billy'`, `'billlly'`, `'biggy'`, `'biggggy'`, etc.

Specifically, it should match any string with a **positive even** number of `'l'`s in the middle, or a **positive even** number of `'g'`s in the middle.

In [ ]:

Intermediate regex

More regex syntax

operation	example	matches ✅	does not match ❌
wildcard	`.U.U.U.`	`'CUMULUS'` `'JUGULUM'`	`'SUCCUBUS'` `'TUMULTUOUS'`
character class	`[A-Za-z][a-z]*`	`'word'` `'Capitalized'`	`'camelCase'` `'4illegal'`

operation	example	matches ✅	does not match ❌
at least one	`bi(ll)+y`	`'billy'` `'billllly'`	`'biy'` `'bily'`

between i and j occurrences	`m[aeiou]{1,2}m`	`'mem'`
		`'maam'`
		`'miem'`
			`'mm'`
			`'mooom'`
			`'meme'`

`.`, `[`, `]`, `+`, `{`, and `}` are also special characters, in addition to `|`, `(`, `)`, and `*`.

***Example (character classes, at least one)**: `[A-E]+` is just shortform for `(A|B|C|D|E)(A|B|C|D|E)`.*

Example (wildcard):

- What does `.` match?
- What does `he.` match?
- What does `...` match?

Example (at least one, closure):

- What does `123+` match?
- What does `123*` match?

Example (number of occurrences): What does `tri{3, 5}` match? Does it match `'triiiii'`?

Example (character classes, number of occurrences): What does `[1-6a-f]{3}-[7-9E-S]{2}` match?

Question 🤔

Write a regular expression that matches any lowercase string has a repeated vowel, such as `'noon'`, `'peel'`, `'festoon'`, or `'zeebraa'`.

In [ ]:

Question 🤔

Write a regular expression that matches any string that contains **both** a lowercase letter and a number, in any order. Examples include `'billy80'`, `'80!!billy'`, and `'bi18ly0'`.

In [ ]:

Even more regex syntax

operation	example	matches ✅	does not match ❌
escape character	`ucsd\.edu`	`'ucsd.edu'`	`'ucsd!edu'`
beginning of line	`^ark`	`'ark two'` `'ark o ark'`	`'dark'`
end of line	`ark$`	`'dark'` `'ark o ark'`	`'ark two'`
zero or one	`cat?`	`'ca'` `'cat'`	`'cart'` (matches `'ca'` only)
built-in character classes*	`\w+` `\d+`	`'billy'` `'231231'`	`'this person'` `'858 people'`
character class negation	`[^a-z]+`	`'KINGTRITON551'` `'1721$$'`	`'porch'` `'billy.edu'`

**Note*: in Python's implementation of regex,

- `\d` refers to digits.
- `\w` refers to alphanumeric characters (`[A-Z][a-z][0-9]_`). **Whenever we say "alphanumeric" in an assignment, we're referring to `\w` !**
- `\s` refers to whitespace.
- `\b` is a word boundary.

Example (escaping):

- What does `he.` match?
- What does `he\.` match?
- What does `(858)` match?
- What does `\(858\)` match?

Example (anchors):

- What does `858-534` match?
- What does `^858-534` match?
- What does `858-534$` match?

Example (built-in character classes):

- What does `\d{3} \d{3}-\d{4}` match?
- What does `\bcat\b` match? Does it find a match in `'my cat is hungry'`?
- What about `'concatenate'` or `'kitty cat'`?

Remember, in Python's implementation of regex,

- `\d` refers to digits.
- `\w` refers to alphanumeric characters (`[A-Z][a-z][0-9]_`). **Whenever we say "alphanumeric" in an assignment, we're referring to `\w`!**
- `\s` refers to whitespace.
- `\b` is a word boundary.

Question 🤔

Write a regular expression that matches any string that:

- is between 5 and 10 characters long, and
- is made up of only vowels (either uppercase or lowercase, including `'Y'` and `'y'`), periods, and spaces.

Examples include `'yoo.ee.IOU'` and `'AI.I oey'`.

In [ ]:

Regex in Python

`re` in Python

The `re` package is built into Python. It allows us to use regular expressions to find, extract, and replace strings.

In [1]:
```
import re
```

`re.search` takes in a string `regex` and a string `text` and returns the location and substring corresponding to the **first** match of `regex` in `text`.

In [2]:
```
re.search('AB*A',
          'here is a string for you: ABBBA. here is another: ABBBBBBBA')
```

Out[2]:
```
<re.Match object; span=(26, 31), match='ABBBA'>
```

`re.findall` takes in a string `regex` and a string `text` and returns a list of all matches of `regex` in `text`. **You'll use this most often.**

In [3]:
```
re.findall('AB*A',
           'here is a string for you: ABBBA. here is another: ABBBBBBBA')
```

Out[3]: ['ABBBA', 'ABBBBBBBA']

`re.sub` takes in a string `regex`, a string `repl`, and a string `text`, and replaces all matches of `regex` in `text` with `repl`.

In [4]:
```
re.sub('AB*A',
       'billy',
       'here is a string for you: ABBBA. here is another: ABBBBBBBA')
```

Out[4]: 'here is a string for you: billy. here is another: billy'

Raw strings

When using regular expressions in Python, it's a good idea to use **raw strings**, denoted by an `r` before the quotes, e.g. `r'exp'`.

In [5]:
```
re.findall('\bcat\b', 'my cat is hungry')
```

Out[5]: []

In [6]:
```
re.findall(r'\bcat\b', 'my cat is hungry')
```

Out[6]: ['cat']

In [7]:
```
# Huh?
print('\bcat\b')
```

```
ca
```

Capture groups

- Surround a regex with `(` and `)` to define a **capture group** within a pattern.
- Capture groups are useful for extracting relevant parts of a string.

In [8]:
```
re.findall(r'\w+@(\w+)\.edu',
           'my old email was billy@notucsd.edu, my new email is notbilly@ucs
```

Out[8]: ['notucsd', 'ucsd']

- Notice what happens if we remove the `(` and `)`!

In [9]:
```
re.findall(r'\w+@\w+\.edu',
           'my old email was billy@notucsd.edu, my new email is notbilly@ucs
```

Out[9]: ['billy@notucsd.edu', 'notbilly@ucsd.edu']

- Earlier, we also saw that parentheses can be used to group parts of a regex together. When using `re.findall`, all groups are treated as capturing groups.

In [10]:
```
# A regex that matches strings with two of the same vowel followed by 3 digi
# We only want to capture the digits, but...
re.findall(r'(aa|ee|ii|oo|uu)(\d{3})', 'eeoo124')
```

Out[10]:
```
[('oo', '124')]
```

Example: Log parsing

Web servers typically record every request made of them in the "logs".

In [11]:
```
s = '''132.249.20.188 - - [24/Feb/2023:12:26:15 -0800] "GET /my/home/ HTTP/1
```

Let's use our new regex syntax (including capturing groups) to extract the day, month, year, and time from the log string `s`.

In [12]:
```
exp = r'\[(.+)\/(.+)\/(.+):(.+):(.+):(.+) .+\]'
re.findall(exp, s)
```

Out[12]:
```
[('24', 'Feb', '2023', '12', '26', '15')]
```

While above regex works, it is not very **specific**. It *works* on incorrectly formatted log strings.

In [13]:
```
other_s = '[adr/jduy/wffsdffs:r4s4:4wsgdfd:asdf 7]'
re.findall(exp, other_s)
```

Out[13]:
```
[('adr', 'jduy', 'wffsdffs', 'r4s4', '4wsgdfd', 'asdf')]
```

The more specific, the better!

- Be as specific in your pattern matching as possible – you don't want to match and extract strings that don't fit the pattern you care about.
  - `.*` matches every possible string, but we don't use it very often.

- A better date extraction regex:

```
\[(\d{2})\/([A-Z]{1}[a-z]{2})\/(\d{4}):(\d{2}):(\d{2}):
(\d{2}) -\d{4}\]

- `\d{2}` matches any 2-digit number.
- `[A-Z]{1}` matches any single occurrence of any uppercase
letter.
- `[a-z]{2}` matches any 2 consecutive occurrences of
lowercase letters.
- Remember, special characters (`[`, `]`, `/`) need to be
d with `\`.
```

In [14]:
```
s
```

Out[14]:
```
'132.249.20.188 - - [24/Feb/2023:12:26:15 -0800] "GET /my/home/ HTTP/1.1" 2
00 2585'
```

In [15]:
```
new_exp = r'\[(\d{2})\/([A-Z]{1}[a-z]{2})\/(\d{4}):(\d{2}):(\d{2}):(\d{2}) -
re.findall(new_exp, s)
```

Out[15]:
```
[('24', 'Feb', '2023', '12', '26', '15')]
```

A benefit of `new_exp` over `exp` is that it doesn't capture anything when the string doesn't follow the format we specified.

In [16]:
```
other_s
```

Out[16]:
```
'[adr/jduy/wffsdffs:r4s4:4wsgdfd:asdf 7]'
```

In [17]:
```
re.findall(new_exp, other_s)
```

Out[17]:
```
[]
```

Question 🤔

`^\w{2,5}.\d*\/[^A-Z5]{1,}`

Select all strings below that contain any match with the regular expression above.

- `"billy4/Za"`
- `"billy4/za"`
- `"DAI_s2154/pacific"`
- `"daisy/ZZZZZ"`
- `"bi_/_lly98"`
- `"!@__!14/atlantic"`

Limitations of regular expressions

Writing a regular expression is like writing a program.

- You need to know the syntax well.
- They can be easier to write than to read.
- They can be difficult to debug.

Regular expressions are terrible at certain types of problems. Examples:

- Anything involving counting (same number of instances of a and b).
- Anything involving complex structure (palindromes).
- Parsing highly complex text structure (HTML, for instance).

Other places where regexes are used

- We've seen how regular expressions are used in Python.
- Regexes show up in a lot of other places, too.
- Example: your favorite text editor's (VSCode, Vim, etc.) search and replace function
- Example: common tools for searching for files, like `grep`
- Example: database queries

LLMs, Regular Expressions, and You

- LLMs are pretty good at writing regexes.
- But beware: there are different "flavors".
  - E.g., Python regular expression syntax/features are *slightly* different from VSCode's which are different from Vim's, etc.
- Tip: be sure to tell the LLM which language/editor/tool you're using.

In my opinion...

- Regular expressions are a powerful tool. But...

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

- Tip: prefer the simple built-in string functions/methods, like `str.contains()`. Only move on to using regular expressions when the added complexity is justified (i.e., the built-in string methods aren't enough).

Summary, next time

Summary

- Regular expressions are used to match and extract patterns from text.
- You don't need to force yourself to "memorize" regex syntax – refer to the resources in the Agenda section of the lecture and on the Resources tab of the course website.
- Also refer to the three tables of syntax in the lecture:
  - Regex building blocks.
  - More regex syntax.
  - Even more regex syntax.
- Note: You don't always have to use regular expressions! If Python/pandas string methods work for your task, you can still use those.
- Play Regex Golf to practice! 🏌️

- `pandas` `.str` methods can use regular expressions; just set `regex=True`.

Next time

- Text features: Bag of words, TF-IDF

In [ ]: