(cartoon)

Pattern Matching with Regular Expressions for Text Processing

"80% of a data analyst's time is spent cleaning up data." `re.search()`, below, finds lines containing data in text and `re.sub()` extracts the data from those lines.

Pattern Matching

```
import re # 're' refers to 'regular expressions'
```

- If `pattern` occurs in `string`, `m = re.search(pattern, string)` returns a *match object* `m` from which we can find the matched string via `m.group()` and the position of the match via `m.start()` and `m.end()`. Otherwise `search()` returns `None`. e.g.

  ```
  m = re.search(pattern='Joe', string='Brown,Joe 123 1000')
  print(f'm.group()={m.group()}, m.start()={m.start()}, m.end()={m.end()}')
  ```

- `re.sub(pattern, repl, string)` returns the string obtained by replacing each occurrence of `pattern` with `repl` ("replacement") in `string`. e.g.

  ```
  re.sub(pattern='1', repl='X', string='Brown,Joe 123 1000')
  ```

Regular Expressions

A *regular expression* describes a set of character strings. In a regular expression,

- letters and digits (`a-z`, `A-Z`, `0-9`) match themselves
- `.` matches any single character
- `\d` matches a *digit* character: `0123456789`

  Note: regular expression escape sequences are written with *one* backslash in python documentation, as in `\d`. But, in a Python string, that one backslash must be typed *twice*, as in `'\\d'`. The first backslash says, "an escape sequence is underway ..." and the second says, "... and the escape sequence is the one for backslash." In a raw string, e.g. `r'\d'`, it is only necessary to type the backslash once.
- `\w` matches a *word* character: a letter, digit, or `_` (underscore)
- `\s` matches a *space* character: space, tab, and newline (and some others)
- `\D`, `\W` and `\S` negate the previous three classes. e.g.

```
re.sub(pattern='a', repl='X', string='abc 012? def 345.')
re.sub(pattern=r'\d', repl='XYZ_', string='abc 012? def 345.')
re.sub(pattern=r'\D', repl='X', string='abc 012? def 345.')
re.sub(pattern=r'\w', repl='X', string='abc 012? def 345.')
re.sub(pattern=r'\W', repl='X', string='abc 012? def 345.')
re.sub(pattern=r'\s', repl='X', string='abc 012? def 345.')
re.sub(pattern=r'\S', repl='X', string='abc 012? def 345.')
```

- square brackets, `[...]`, enclose a *character class* that matches any one of its characters; except that `[^...]` matches any one character *not* in the class; e.g.

```
re.sub(pattern='[aeiou]' , repl='', string='abc 012? def 345.') # strip vowels
re.sub(pattern='[^aeiou]', repl='', string='abc 012? def 345.') # strip non-vowels
```

- `^` matches the beginning of a line (`$` matches the end); e.g.

```
re.sub(pattern='^a', repl='X', string='abc abc')
re.sub(pattern='c$', repl='X', string='abc abc')
```

- `\b` matches the empty string at a word boundary; and `\B` matches the empty string not at a boundary e.g.

```
re.sub(pattern=r'c\b', repl='X', string='abc abcd abc')
re.sub(pattern=r'c\B', repl='X', string='abc abcd abc')
```

- repetition quantifiers in `{...}` indicate matching the previous expression
  - `{n}` exactly `n` times
  - `{n, }` `n` or more times (shorthand: `*` means `{0, }`, `+` means `{1, }`)
  - `{n,m}` `n` to `m` times (shorthand: `?` means `{0,1}` or "optional"); e.g.

```
strings = ['abc 012', 'abc 0123', 'def 456', 'def 4567']
[re.sub(r'\d{2}$', 'X', s) for s in strings] # 2 digits, end-of-line
[re.sub(r' \d{2}$', 'X', s) for s in strings] # space, 2 digits, end-of-line
```

Here is a pattern that is helpful for the quiz (and unimportant otherwise). It counts the strings with space, 2 or 3 digits, end-of-line: `len([s for s in strings if re.search(' \d{2,3}$', s)])`

Note: repetition is maximal, except that appending `?` to a quantifier makes it minimal. e.g.

```
[re.sub('\\d{1,}', 'X', s) for s in strings] # also try '?' after '}'
```

- parentheses in the `pattern` enclose an expression; a *backreference* `\N` (where `N` is in `1` to `9`) in the replacement string `repl` refers to what the N^{th} enclosed expression matched; e.g.

```
link = 'blah blah blah ... <a href=http://www.google.com>Google</a> blah ...'
re.sub(pattern='.*<a href=(.*)>.*'   , repl=r'\1', string=link) # match too much
re.sub(pattern='.*<a href=(.*?)>.*'  , repl=r'\1', string=link) # one fix
re.sub(pattern='.*<a href=([^>]*)>.*', repl=r'\1', string=link) # another fix

# rewrite 'last,first ID email ...' to '.csv': 'first,last,user,ID'
  a = ['Brown,Joe    123456789 jbrown@wisc.edu      1000',
    'Roukos,Sally 456789123 sroukos@wisc.edu     5000',
    'Chen,Jean    789123456 chen@wisc.edu       24000',
    'Juniper,Jack 345678912 jjuniper@wisc.edu  300000']
  [re.sub(pattern=r'(\w+),(\w+) +(\d+) (\w+).*', repl=r'\2,\1,\4,\3', string=s)
    for s in a]
```

Note: It is frequently useful, as in this preceding example, to match the whole line to ensure that the context is correct for the part of the match you care about.

Backreferences in the replacement string are the best feature of regular expressions.

- `|` means *or*; e.g.

  ```
  [s for s in a if re.search(pattern='Joe|Jack', string=s)]
  ```

  ```
  [s for s in a if re.search(pattern='J(o|a)', string=s)]
  ```

- `. \ | ( ) [ { ^ $ * + ?` are *metacharacters* with special meaning; to use them as regular characters, *escape* them with `\` (doubled if not in a raw string, as described above)

Splitting Strings

`re.split(pattern, string)` splits `string` on `pattern`, returning a list of substrings.

```
[re.split(pattern=',', string=s) for s in a] # split on comma, miss split on whitespace
[re.split(pattern=' +', string=s) for s in a] # split on whitespace, miss split on comma
[re.split(pattern=r',|\( +\)', string=s) for s in a] # split on either: good
```

To learn more, see Regular Expression HOWTO and Regular expression operations.

Friendly advice: Budget for a headache. Re-read the "80%" line. Memorize the cartoon hero's line.