Regular Expression History Revisited

- You've seen two versions of Ken Thompson's regex language:
  - POSIX **Basic Regular Expressions**
    - limited syntax, e.g no `|`
    - used by `grep` & `sed`
    - needed when computers were every slow to make regex matching faster
  - POSIX **Extended Regular Expressions** - superset of Basic Regular Expressions
    - used by **`grep -E`** & **`sed -E`**
- Henry Spencer produced the first open source regex library
  - used many place e.g. postgresql, tcl
  - extended (added features & syntax) to Ken's regex language.
- Perl (Larry Wall) copied Henry's library & extended much further
  - available outside Perl via **Perl Compatible Regular Expressions** library
  - used by **`grep -P`**
- Python standard **`re`** package also copied Henry's library
  - added most of the features in Perl/PCRE
  - many commonly used features are common to both
- we will cover useful regex features added by Python & Perl/PCRE
- https://regex101.com/ lets you specify which regex language

Python **re** package - useful functions

```
re.search(regex, string, flags)
```

- search for a *regex* match within *string*
- return object with information about match or None if match fails
- optional parameter modifies matching, e.g. make matching case-insensitive with: `flags=re.I`

```
re.match(regex, string, flags)
```

- only match at start of string
- same as `re.search` stating with ^

```
re.fullmatch(regex, string, flags)
```

- only match the full string
- same as `re.search` stating with ^ and ending with $

Python **re** package - useful functions

```
re.sub(regex, replacement, string, count, flags)
```

- return *string* with anywhere *regex* matches, substituted by *replacement*
- optional parameter *count*, if non-zero, sets maximum number of substitutions

```
re.findall(regex, string, flags)
```

- return all non-overlapping matches of pattern in string

```
re.split(regex, string, maxsplit, flags)
```

- Split *string* everywhere *regex* matches
- optional parameter *maxsplit*, if non-zero, set maximum number of splits

Character Classes

Python (& PCRE) regular expression adds character classes

`\d`	matches any *digit*, for ASCII: `[0-9]`
`\D`	matches any non-*digit*, for ASCII: `[^0-9]`
`\w`	matches any *word* char, for ASCII: `[a-zA-Z_0-9]`
`\W`	matches any non-*word* char, for ASCII: `[^a-zA-Z_0-9]`
`\s`	matches any *whitespace*, for ASCII: `[ \t\n\r\f]`
`\S`	matches any non-*whitespace*, for ASCII: `[^ \t\n\r\f]`
`\b`	matches at a word boundary
`\B`	matches except at a word boundary
`\A`	matches at the start of the string, same as **^**
`\Z`	matches at the end of the string, same as **$**

- convenient
- make your regex more likely to be portable to non-English locales
- **\b** and **\B** are like **^** and **$** - they don't match characters, they anchor the match

raw strings

- Python raw-string is prefixed with an r (for raw)
  - can prefix with r strings quoted with `'` `"` `'''` `"""`
- backslashes have no special meaning in raw-string except before quotes
  - backslashes escape quotes but also stay in the string
- regexes often contain backslashes - using raw-strings makes them more readable

```
>>> print('Hello\nAndrew')
Hello
Andrew
>>> print(r'Hello\nAndrew')
Hello\nAndrew
>>> r'Hello\nAndrew' == 'Hello\\nAndrew'
True
>>> len('\n')
1
>>> len(r'\n')
2
```

Match objects

- `re.search`, `re.match`, `re.fullmatch` return a match object if a match suceeds, None if it fails
  - hence their return can to control `if` or `while`

```
print("Destroy the file system? ")
answer = input()
if re.match(r'yes|ok|affirmative', answer, flags=re.I):
   subprocess.run("rm -r /", Shell=True)
```

- the match object can provide useful information:

```
>>> m = re.search(r'[aiou].*[aeiou]', 'pillow')
>>> m
<re.Match object; span=(1, 5), match='illo'>
>>> m.group(0)
'illo'
>>> m.span()
(1, 5)
>>>
```

Capturing Parts of a Regex Match

- brackets are used for grouping (like arithmetic) in extened regular expresions
- in Python (& PCRE) brackets also capture the part of the string matched
- **`group(`*n*`)`** returns part of the string matched by the ***n***th-pair of brackets

```
>>> m = re.search('(\w+)\s+(\w+)', 'Hello Andrew')
>>> m.groups()
('Hello', 'Andrew')
>>> m.group(1)
'Hello'
>>> m.group(2)
'Andrew'
```

- **`\number`** can be used to refer to group *number* in an `re.sub` replacement string

```
>>> re.sub(r'(\d+) and (\d+)', r'\2 or \1', "The answer is 42 and 43?")
'The answer is 43 or 42?'
```

Back-referencing

- **`\number`** can be used further on in a regex - often called a back-reference
  - e.g. **`r'^(\d+) (\1)$'`** match the same integer twice

```
>>> re.search(r'^(\d+) (\d+)$', '42 43')
<re.Match object; span=(0, 5), match='42 43'>
>>> re.search(r'^(\d+) (\1)$', '42 43')
>>> re.search(r'^(\d+) (\1)$', '42 42')
<re.Match object; span=(0, 5), match='42 42'>
```

- back-references allow matching impossible with classical regular expressions
- python supports up to 99 back-references, `\1`, `\2`, `\3`, ..., `\99`
  - `\01` or `\100` is interpreted as an octal number

Non-Capturing Group

- `(?:...)` is a non-capturing group
  - it has the same grouping behaviour as `(...)`
  - it doesn't capture the part of the string matched by the group

```
>>> m = re.search(r'.*(?:[aeiou]).*([aeiou]).*', 'abcde')
>>> m
<re.Match object; span=(0, 5), match='abcde'>
>>> m.group(1)
'e'
```

Greedy versus non-Greedy Pattern Matching

- The default semantics for pattern matching is **greedy**:
  - starts match the first place it can succeed
  - make the match as long as possible
- The **?** operator changes pattern matching to **non-greedy**:
  - starts match the first place it can succeed
  - make the match as short as possible

```
>>> s = "abbbc"
>>> re.sub(r'ab+', 'X', s)
'Xc'
>>> re.sub(r'ab+?', 'X', s)
'Xbbc'
```

Why Implementing a Regex Matching isn't Easy

- regex matching starts match the first place it can succeed
- but a regex can partly match many places

```
>>> re.sub(r'ab+c', 'X', "abbabbbbbbbabbbc")
'abbabbbbbbbX'
```

- and may need to ***backtrack***, e.g:

```
>>> re.sub(r'a.*bc', 'X', "abbabbbbbbbcabbb")
'Xabbb'
```

- poorly design regex engines can get very slow
  - have been used for denial-of-service attacks
- Python extensions (back-references) make matching **NP-hard**

re.findall

- `re.findall` returns a list of the matched strings, e.g:

```
>>> re.findall(r'\d+', "-5==10zzz200_")
['5', '10', '200']
```

- if the regex contains `()` only the captured text is returned

```
>>> re.findall(r'(\d)\d*', "-5==10zzz200_")
['5', '1', '2']
```

- if the regex contains multiple `()` a list of tuples is return

```
>>> re.findall(r'(\d)\d*(\d)', "-5==10zzz200_")
[('1', '0'), ('2', '0')]
>>> re.findall(r'([^,]*), (\S+)', "Hopper, Grace Brewster Murray")
[('Hopper', 'Grace')]
>>> re.findall(r'([A-Z])([aeiou])', "Hopper, Grace Brewster Murray")
[('H', 'o'), ('M', 'u')]
```

re.split

- `re.split` splits a string where a regex match

```
>>> re.split(r'\d+', "-5==10zzz200_")
['-', '==', 'zzz', '_']
```

- like `cut` in Shell scripts - but more powerful
- for example, you can't do this with `cut`

```
>>> re.split(r'\s*,\s*', "abc,de, ghi    ,jk    , mn")
['abc', 'de', 'ghi', 'jk', 'mn']
```

see also the string `join` function

```
>>> a = re.split(r'\s*,\s*', "abc,de, ghi    ,jk    , mn")
>>> a
['abc', 'de', 'ghi', 'jk', 'mn']
>>> ':'.join(a)
'abc:de:ghi:jk:mn'
```

Example - When Harry Met Hermione #0

```python
# For each file given as argument replace occurrences of Hermione
# allowing for some misspellings with Harry and vice-versa.
# Relies on Zaphod not occurring in the text.
import re, sys, os
for filename in sys.argv[1:]:
    tmp_filename = filename + ".new"
    if os.path.exists(tmp_filename):
        print(f"{sys.argv[0]}: {tmp_filename} already exists\n", file=sys.stderr)
        sys.exit(1)
    with open(filename) as f:
        with open(tmp_filename, "w") as g:
            for line in f:
                changed_line = re.sub(r"Herm[io]+ne", "Zaphod", line)
                changed_line = changed_line.replace("Harry", "Hermione")
                changed_line = changed_line.replace("Zaphod", "Harry")
                g.write(changed_line)
    os.rename(tmp_filename, filename)
```

source code for trans.0.py

Example - When Harry Met Hermione #1

```
# For each file given as argument replace occurrences of Hermione
# allowing for some misspellings with Harry and vice-versa.
# Relies on Zaphod not occurring in the text.
import re, sys, os, shutil, tempfile
for filename in sys.argv[1:]:
    with tempfile.NamedTemporaryFile(mode='w', delete=False) as tmp:
        with open(filename) as f:
            for line in f:
                changed_line = re.sub(r"Herm[io]+ne", "Zaphod", line)
                changed_line = changed_line.replace("Harry", "Hermione")
                changed_line = changed_line.replace("Zaphod", "Harry")
                tmp.write(changed_line)
    shutil.move(tmp.name, filename)
```

source code for trans.1.py

Example - When Harry Met Hermione #2

```
# For each file given as argument replace occurrences of Hermione
# allowing for some misspellings with Harry and vice-versa.
# Relies on Zaphod not occurring in the text.
# modified text is stored in a list then file over-written
import re, sys, os
for filename in sys.argv[1:]:
    changed_lines = []
    with open(filename) as f:
        for line in f:
            changed_line = re.sub(r"Herm[io]+ne", "Zaphod", line)
            changed_line = changed_line.replace("Harry", "Hermione")
            changed_line = changed_line.replace("Zaphod", "Harry")
            changed_lines.append(changed_line)
    with open(filename, "w") as g:
        g.write("".join(changed_lines))
```

source code for trans.2.py

Example - When Harry Met Hermione #3

```
# For each file given as argument replace occurrences of Hermione
# allowing for some misspellings with Harry and vice-versa.
# Relies on Zaphod not occurring in the text.
# modified text is stored in a single string then file over-written
import re, sys, os
for filename in sys.argv[1:]:
    changed_lines = []
    with open(filename) as f:
        text = f.read()
    changed_text = re.sub(r"Herm[io]+ne", "Zaphod", text)
    changed_text = changed_text.replace("Harry", "Hermione")
    changed_text = changed_text.replace("Zaphod", "Harry")
    with open(filename, "w") as g:
        g.write("".join(changed_text))
```

source code for trans.3.py

Example - printing the last number

```python
# Print the last number (real or integer) on every line
# Note: regexp to match number:  -?\d+\.?\d*
# Note: use of assignment operator :=
import re, sys
for line in sys.stdin:
    if m := re.search(r'(-?\d+\.?\d*)\D*$', line):
        print(m.group(1))
```

source code for print_last_number.py

Example - finding numbers #0

```
# Find the positive integers among input text
# print their sum and mean
# Note regexp to split on non-digits
# Note check to handle empty string from split
# Only positive integers handled
import re, sys
input_as_string = sys.stdin.read()
numbers = re.split(r"\D+", input_as_string)
print(numbers)
total = 0
n = 0
for number in numbers:
    if number:
        total += int(number)
        n += 1
if numbers:
    print(n, "numbers: total", total, "with mean", total / n)
```

source code for find_numbers.0.py

Example - finding numbers #1

```
# Find the positive integers among input text
# print their sum and mean
# Note regexp to match number -?\d+\.?\d*
# match postive & integers & floating-point numbers
import re, sys
input_as_string = sys.stdin.read()
numbers = re.findall(r"-?\d+\.?\d*", input_as_string)
print(numbers)
n = len(numbers)
total = sum(float(number) for number in numbers)
if numbers:
    print(n, "numbers: total", total, "with mean", total / n)
```

source code for find_numbers.1.py