Regular Expression History Revisited

- 1950s mathematician Stephen Kleene develops theory
- 1960s Ken Thompson develops syntax and practical implementation, two versions:
  - POSIX Basic Regular Expressions
    - limited syntax, e.g no `|`
    - used by `grep` & `sed`
    - needed when computers were very slow to make regex matching faster
  - POSIX Extended Regular Expressions - superset of Basic Regular Expressions
    - used by `grep -E` & `sed -E`
- 1980s Henry Spencer produces open source regex C library
  - used many place e.g. postgresql, tcl
  - extended (added features & syntax) to Ken’s regex language.
- 1987 Perl (Larry Wall) copied Henry’s library & extended much further
  - available outside Perl via Perl Compatible Regular Expressions library
  - used by `grep -P`
- 1990s 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 some (not all) useful extra regex features found in both Python & Perl/PCRE
- note https://regex101.com/ lets you specify which regex language

Python re package - useful functions

```python
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 flags modifies matching,
# e.g. make matching case-insensitive with: `flags=re.I`
```
```python
re.match(regex, string, flags)
# only match at start of string
# same as `re.search` stating with `^`
```
```python
re.fullmatch(regex, string, flags)
# only match the full string
# same as `re.search` stating with `^` and ending with `$`
```
Python `re` package - useful functions

```python
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
# if pattern contains () return part matched by ()
# if pattern contains multiple () return tuple

re.split(regex, string, maxsplit, flags)
# Split *string* everywhere *regex* matches
# optional parameter *maxsplit*, if non-zero, set maximum number of splits
```

Python Characters Classes (also in PCRE)

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>ASCII Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>matches any digit</td>
<td>[0-9]</td>
</tr>
<tr>
<td>\D</td>
<td>matches any non-digit</td>
<td>^[^0-9]</td>
</tr>
<tr>
<td>\w</td>
<td>matches any word char</td>
<td>[a-zA-Z_0-9]</td>
</tr>
<tr>
<td>\W</td>
<td>matches any non-word char</td>
<td>^[^a-zA-Z_0-9]</td>
</tr>
<tr>
<td>\s</td>
<td>matches any whitespace, for ASCII:</td>
<td>[ \t\n\r\f]</td>
</tr>
<tr>
<td>\S</td>
<td>matches any non-whitespace, for ASCII:</td>
<td>^[\t\n\r\f]</td>
</tr>
<tr>
<td>\b</td>
<td>matches at a word boundary</td>
<td>^</td>
</tr>
<tr>
<td>\B</td>
<td>matches except at a word boundary</td>
<td>^</td>
</tr>
<tr>
<td>\A</td>
<td>matches at the start of the string, same as ^</td>
<td></td>
</tr>
<tr>
<td>\Z</td>
<td>matches at the end of the string, same as $</td>
<td></td>
</tr>
</tbody>
</table>

- convenient and 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

```python
>>> print('Hello
Andrew')
Hello
Andrew
>>> print(r'Hello
Andrew')
Hello\nAndrew
>>> r'Hello
Andrew' == '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 succeeds, None if it fails
  - hence their return can control `if` or `while`

```python
def main():
    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:

```python
>>> m = re.search(r'[aiou].*[aeiou]', 'pillow')
>>> m
<re.Match object; span=(1, 5), match='illo'>
>>> m.group(0)
'illo'
>>> m.span()
(1, 5)
>>> m.groups()
('pill',)
>>> m.group(1)
'pill'
>>> m.group(2)
''
```

Capturing Parts of a Regex Match

- brackets are used for grouping (like arithmetic) in extended regular expressions
- `group(n)` returns part of the string matched by the `n`th-pair of brackets

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

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

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

Back-referencing

- `
umber` can also be used in a regex as well
- usually called a back-reference
  - e.g. `r'\^\d+ (\1)$'` match the same integer twice

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

```python
>>> 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

```python
>>> 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

```python
>>> re.sub(r'ab+c', 'X', "abbabbbbbbabbbc")
'abbabbbbbbbX'
```
- and may need to **backtrack**, e.g:

```python
>>> re.sub(r'a.*bc', 'X', "abbabbbbbcabbb")
'Xabb'
```
- poorly designed regex engines can get very slow
  - have been used for denial-of-service attacks
- Python (PCRE) regex matching is **NP-hard** due to back-references
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]*)', "-5==10zzz200_")
  ['5', '10', '200']

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

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

>>> re.findall(r'([^aeiou])', "Hopper, Grace Brewster Murray")
[('H', 'o'), ('r', 'a'), ('G', 'r'), ('e', 's'), ('B', 'w'), ('M', 'u')]

re.split

- re.split splits a string where a regex matches

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

# print the sum and mean of any positive integers found on stdin
# Note regexp to split on non-digits
# Only positive integers handled
import re, sys
input_as_string = sys.stdin.read()
numbers = re.split(r"\D+", input_as_string)
total = 0
n = 0
for number in numbers:
    if number:
        total += int(number)
        n += 1
if numbers:
    print(f"{n} numbers, total {total}, mean {total / n:.1f}")

source code for find_numbers.0.py

Example - finding numbers #1

# print the sum and mean of any numbers found on stdin
# Note regexp to match number -?\d+\.?\d*
# match positive & negative integers & floating-point numbers
import re, sys
input_as_string = sys.stdin.read()
numbers = re.findall(r"-?\d+\.?\d\*", input_as_string)
n = len(numbers)
total = sum(float(number) for number in numbers)
if numbers:
    print(f"{n} numbers, total {total}, mean {total / n:.1f}")

source code for find_numbers.1.py

Example - counting enrollments with regexes & dicts

course_names = {}
with open(COURSE_CODES_FILE, encoding="utf-8") as f:
    for line in f:
        if m := re.match(r"([\S]+)\s+\((.*\S)\)", line):
            course_names[m.group(1)] = m.group(2)
enrollments_count = {}
with open(ENROLLMENTS_FILE, encoding="utf-8") as f:
    for line in f:
        course_code = re.sub(r"\|.*\n", "", line)
        if course_code in enrollments_count:
            enrollments_count[course_code] += 1
        else:
            enrollments_count[course_code] = 0

for (course_code, enrollment) in sorted(enrollments_count.items()):
    if course_code not in enrollments_count:
        name = course_names.get(course_code, "???")
        print(f"{enrollment:4} {course_code} {name}"
**Example - counting enrollments with split & counters**

```python
course_names = {}
with open(COURSE_CODES_FILE, encoding="utf-8") as f:
  for line in f:
    course_code, course_name = line.strip().split("\t", maxsplit=1)
    course_names[course_code] = course_name

enrollments_count = collections.Counter()
with open(ENROLLMENTS_FILE, encoding="utf-8") as f:
  for line in f:
    course_code = line.split("|")[0]
    enrollments_count[course_code] += 1

for (course_code, enrollment) in sorted(enrollments_count.items()):
  # if no name for course_code use ???
  name = course_names.get(course_code, "???")
  print(f"{enrollment:4} {course_code} {name}")
```

source code for count_enrollments.py

**Example - counting first names**

```python
already_counted = set()
first_name_count = collections.Counter()
with open(ENROLLMENTS_FILE, encoding="utf-8") as f:
  for line in f:
    _, student_number, full_name = line.split("|")[:3]
    if student_number in already_counted:
      continue
    already_counted.add(student_number)
    if m := re.match(r".*,\s+([^,]+)\", full_name):
      first_name = m.group(1)
      first_name_count[first_name] += 1

# put the count first in the tuples so sorting orders on count before name
count_name_tuples = [(c, f) for (f, c) in first_name_count.items()]
# print first names in decreasing order of popularity
for (count, first_name) in sorted(count_name_tuples, reverse=True):
  print(f"{count:4} {first_name}")
```

source code for count_first_names.py

**Example - finding duplicate first names using dict of dicts**

```python
course_first_name_count = {}
with open(ENROLLMENTS_FILE, encoding="utf-8") as f:
  for line in f:
    course_code, _, full_name = line.split("|")[:3]
    if m := re.match(r".*,\s+([^,]+)\", full_name):
      first_name = m.group(1)
      course_first_name_count[course_code] = {}
      course_first_name_count[course_code][first_name] = 0
      course_first_name_count[course_code][first_name] += 1
  for course in course_first_name_count:
    for (first_name, count) in course_first_name_count[course].items():
      if count >= REPORT_MORE_THAN_STUDENTS:
        print(f"{course}, "has", count, "students named", first_name)
```

source code for duplicate_first_names.py
Example - finding duplicate first names using split & defaultdict of counters

```python
course_first_name_count = collections.defaultdict(collections.Counter)
with open(ENROLLMENTS_FILE, encoding="utf-8") as f:
    for line in f:
        course_code, _, full_name = line.split"|\)[0:3]
given_names = full_name.split"\,"[1].strip()
first_name = given_names.split\(\ )[0]
course_first_name_count\[course_code\][first_name] += 1

for (course, name_counts) in sorted(course_first_name_count.items()):
    for (first_name, count) in name_counts.items():
        if count > REPORT_MORE_THAN_STUDENTS:
            print(course, "has", count, "students named", first_name)
```

Example - Changing Filenames with Regex

```python
# written by andrewt@unsw.edu.au for COMP(2041|9044)
#
# Change the names of the specified files
# by substituting occurrences of regex with replacement
# (simple version of the perl utility rename)
import os
import re
import sys
if len(sys.argv)< 3:
    print(f"Usage:{sys.argv[0]} <regex> <replacement> [files]", file=sys.stderr)
    sys.exit(1)
regex= sys.argv[1]
replacement= sys.argv[2]
for old_pathname in sys.argv[3:]:
    new_pathname= re.sub(regex, replacement, old_pathname, count=1)
    if new_pathname==old_pathname:
        continue
    if os.path.exists(new_pathname):
        print(f"{sys.argv[0]}: '{new_pathname}' exists", file=sys.stderr)
        continue
    try:
        os.rename(old_pathname, new_pathname)
        except
            print(f"{sys.argv[0]}: '{new_pathname}'{e}", file=sys.stderr)
```

Example - Changing Filenames with Regex & Eval

```python
import argparse
import os
import re
import sys
parser=argparse.ArgumentParser()
# add required arguments
parser.add_argument("regex", type=str, help="match against filenames")
parser.add_argument("replacement", type=str, help="replaces matches with this")
parser.add_argument("filenames", nargs="*", help="filenames to be changed")
# add some optional boolean arguments
parser.add_argument("-d", "--dryrun", action="store_true", help="show changes but don't make")
parser.add_argument("-n", "--replace_n_matches", type=int, default=1, help="replace n matches (0 for all matches)")
args=parser.parse_args()
def eval_replacement(match):
    """if --eval given, evaluate replacement string as Python with the variable _ set to the matching part of the filename"
    if not args.eval:
        return
    args.replacement_ = match.group(0)
    return
str(eval(args.replacement))
for old_pathname in args.filenames:
    try:
        new_pathname= re.sub(args.regex, eval_replacement, old_pathname, count=args.replace_n_matches)
        except
            print(f"{sys.argv[0]}: '{old_pathname}': '{args.replacement}'{e}", file=sys.stderr)
        continue
    if new_pathname==old_pathname:
        if args.verbose:print("no change:", old_pathname)
        continue
    if os.path.exists(new_pathname):
        print(f"{sys.argv[0]}: '{new_pathname}' exists", file=sys.stderr)
        continue
    if args.dryrun:print(old_pathname, "would be renamed to", new_pathname)
    continue
    if args.verbose:print("'renaming", old_pathname,"to", new_pathname)
    try:
        os.rename(old_pathname, new_pathname)
        except
            print(f"{sys.argv[0]}: '{new_pathname}'{e}", file=sys.stderr)
```
# 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 change_names.0.py
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

```python
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 change_names.3.py