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
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
- 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 (& PCRE) regular expression adds character classes

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
<th>ASCII Range</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</td>
<td>[ \t\n\r\f]</td>
</tr>
<tr>
<td>\S</td>
<td>matches any non-whitespace</td>
<td>[^\t\n\r\f]</td>
</tr>
<tr>
<td>\b</td>
<td>matches at a word boundary</td>
<td>[^\t\n\r\f]</td>
</tr>
<tr>
<td>\B</td>
<td>matches except at a word boundary</td>
<td>[^\t\n\r\f]</td>
</tr>
<tr>
<td>\A</td>
<td>matches at the start of the string</td>
<td>^</td>
</tr>
<tr>
<td>\Z</td>
<td>matches at the end of the string</td>
<td>$</td>
</tr>
</tbody>
</table>

- 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
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
Andrew
>>> 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 to control `if` or `while`

```python
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)
```
Capturing Parts of a Regex Match

- brackets are used for grouping (like arithmetic) in extended regular expressions
- 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

```python
>>> 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

```python
>>> 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

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

```python
>>> 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
(?:...)

- (?:...) 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'
```
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', "abbabbbbbbbabbbbc")
'abbabbbbbbbX'
```

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

```python
>>> 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:

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

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

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

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

```python
>>> re.findall(r'\(\d)\d*\(\d\)', "-5==10zzz200_")
[('1', '0'), ('2', '0')]
```

```python
>>> re.findall(r'\([^,]*\), (\S+)', "Hopper, Grace Brewster Murray")
[('Hopper', 'Grace')]
```

```python
>>> 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

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

- like `cut` in Shell scripts - but more powerful

- for example, you can’t do this with `cut`

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

see also the string `join` function

```python
>>> 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

# 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

https://www.cse.unsw.edu.au/~cs2041/22T2/
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.py

https://www.cse.unsw.edu.au/~cs2041/22T2/
# Find the positive integers among input text
# print their sum and mean
# Note regexp to match number -?\d+\.?\d*
# match positive & 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)
import os
import sys

for old_pathname in sys.argv[1:]:
    new_pathname = old_pathname.lower()
    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 OSError as e:
        print(f"{sys.argv[0]}: '{new_pathname}' {e}"", file=sys.stderr)
Example - Changing Filenames with Regex

# 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 OSError as e:
        print(f"{sys.argv[0]}: '{new_pathname}' {e}" , file=sys.stderr)
Example - Changing Filenames with Regex & Eval

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 them")

# optional integer argument which defaults to 1
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 OSError as e:
        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)
    else:
        try:
            os.rename(old_pathname, new_pathname)
        except OSError as e:
            print(f"{sys.argv[0]}: '{new_pathname}'{e}", file=sys.stderr)

source code for rename_regex_eval.py
https://www.cse.unsw.edu.au/~cs2041/22T2/
# 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.

```python
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_names0.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.

```python
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 change_names1.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 list then file over-written

```python
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 change_names2.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