Advanced Python for JPMC Supplement

John Strickler

Version 1.0, April 2023

Table of Contents

Chapter 1: Regular Expressions

Objectives

- Using RE patterns
- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding option flags to a pattern
- Specifying capture groups
- Replacing text with backrefs and callbacks

Regular expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by **Perl**
- Python adds some conveniences

> Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the re module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as Does this string match the pattern?'', or Is there a match for the pattern anywhere in this string?''. You can also use REs to modify a string or to split it apart in various ways.
>
> — Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of **Perl** that they substantially changed from the originals. **Perl** added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses **Perl**-style regular expressions (AKA PCREs) and adds a few extensions of its own.

Two good web sites for creating and deciphering regular expressions:

Regex 101: https://regex101.com/#python
Pythex: http://www.pythex.org/

Two sites for having fun (yes, really!) with regular expressions:

Regex Golf: https://alf.nu/RegexGolf
Regex Crosswords: https://regexcrossword.com/

RE syntax overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more *branches* separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of *atoms*. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a *quantifier* (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.

TIP There is frequently only one branch.

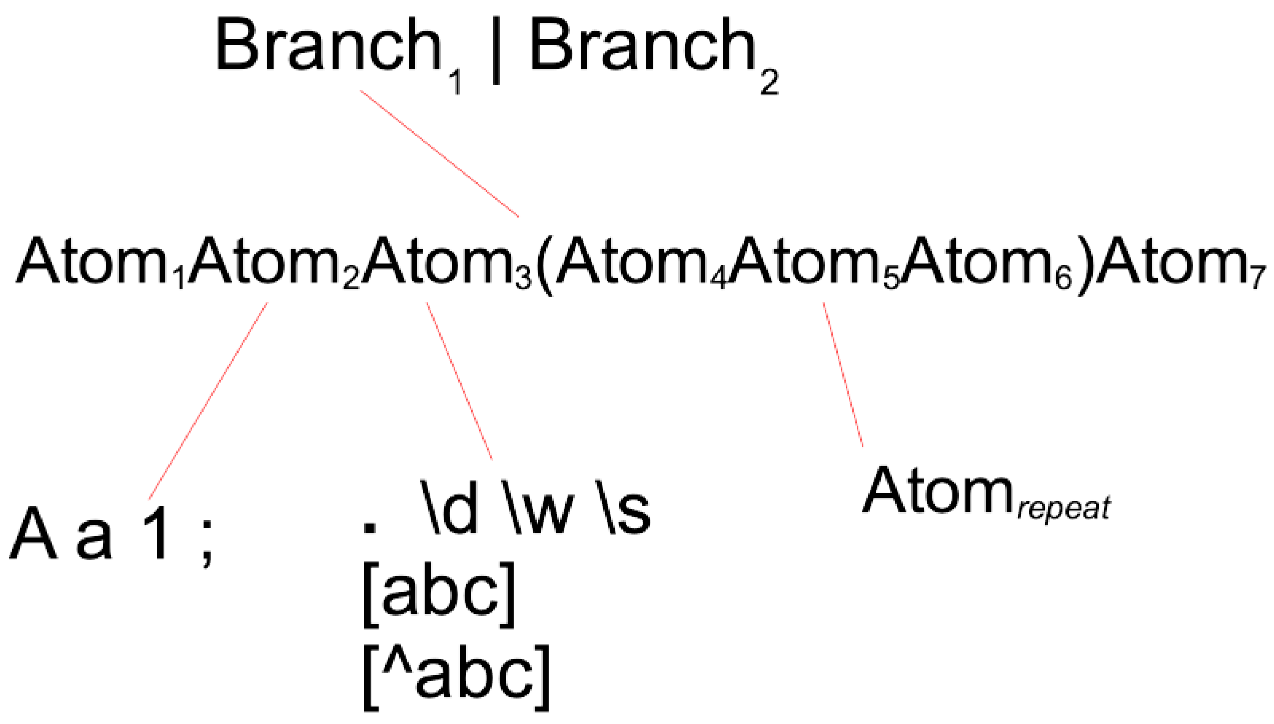

Table 1. Regular Expression Metacharacters

Pattern	Description
`.`	any character
`[abc]`	any character in set
`[^abc]`	any character not in set
`\w,\W`	any word, non-word char
`\d,\D`	any digit, non-digit
`\s,\S`	any space, non-space char
`^,$`	beginning, end of string
`\b`	beginning or end of word
`\`	escape a special character
`*,+,?`	0 or more, 1 or more, 0 or 1
`{m}`	exactly m occurrences
`{m,}`	at least m occurrences
`{m,n}`	m through n occurrences
`a|b`	match a or b
`(?aiLmsux)`	Set the A, I, L, M, S, U, or X flag for the RE (see below).
`(?:⋯)`	Non-capturing version of regular parentheses.
`(?P<name>⋯)`	The substring matched by the group is accessible by name.
`(?P=name)`	Matches the text matched earlier by the group named name.
`(?#⋯)`	A comment; ignored.
`(?=⋯)`	Matches if … matches next, but doesn’t consume the string.
`(?!⋯)`	Matches if … doesn’t match next.
`(?⇐⋯)`	Matches if preceded by … (must be fixed length).
`(?<!⋯)`	Matches if not preceded by … (must be fixed length).

Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

Find first match

```
re.search(pattern, string)
```

Searches s and returns the first match. Returns a match object (**SRE_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

Find all matches

```
re.finditer(pattern, string)
```

Provides a match object for each match found. Normally used with a **for** loop.

Retrieve text of all matches

```
re.findall(pattern, string)
```

Finds all matches and returns a list of matched strings. Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

Other search methods

`re.match()` is like `re.search()`, but searches for the pattern at beginning of `s`. There is an implied `^` at the beginning of the pattern.

Likewise `re.fullmatch()` only succeeds if the pattern matches the entire string. `^` and `$` around the pattern are implied.

Example

regex_finding_matches.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'  # store pattern in raw string

if re.search(pattern, s):  # search returns True on match
    print("Found pattern.")
print()

m = re.search(pattern, s)  # search actually returns match object
print(m)
if m:
    print("Found:", m.group(0))  # group(0) returns text that was matched by entire
expression (or just m.group())
print()

for m in re.finditer(pattern, s):  # iterate over all matches in string:
    print(m.group())
print()

matches = re.findall(pattern, s)  # return list of all matches
print("matches:", matches)
```

regex_finding_matches.py

```
Found pattern.

<re.Match object; span=(12, 17), match='M-302'>
Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

RE objects

- `re` object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An `re` object is created by calling `re.compile()` with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. `re.compile()` has an optional argument for flags which enable special features or fine-tune the match.

TIP It is generally a good practice to create your re objects in a location near the top of your script, and then use them as necessary

Example

regex_objects.py

```python
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]-\d{2,3}')  # Create an re (regular expression) object

if rx_code.search(s):  # Call search() method from the object
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

regex_objects.py

```
Found pattern.

Found: M-302

M-302
H-476
Q-51
A-110
H-332
Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

Compilation flags

- Fine-tune match
- Add readability

When using functions from `re`, or when compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined with `|`, the bitwise ***OR*** operator. Each flag has a short for and a long form.

Ignoring case

```
re.I, re.IGNORECASE
```

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the LOCALE flag.

Using the locale

```
re.L, re.LOCALE
```

Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write \w+ to match words, but \w only matches the character class `[A-Za-z]`; it won't match "é" or "ç". If your system is configured properly and a French locale is selected, certain C functions will tell the program that "é" should also be considered a letter. Setting the `LOCALE` flag enables \w+ to match French words as you'd expect.

Ignoring whitespace

```
re.X, re.VERBOSE
```

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a # that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

Example

regex_flags.py

```python
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'

if re.search(pattern, s, re.IGNORECASE):  # make search case-insensitive
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I | re.M)  # short version of flag
if m:
    print("Found:", m.group())
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

regex_flags.py

```
Found pattern.

Found: M-302

M-302
r-99
H-476
Q-51
z-883
A-110
H-332
Y-45

matches: ['M-302', 'r-99', 'H-476', 'Q-51', 'z-883', 'A-110', 'H-332', 'Y-45']
```

Working with newlines

- `re.SINGLELINE` lets `.` match newline
- `re.MULTILINE` lets `^` and `$` match lines

Some text contains newlines (`\n`), representing mutiple lines within the string. There are two regular expression flags you can use with `re.search()` and other functions to control how they are searched. These flags are not useful if the string has no embedded newlines.

Treating text like a single string

By default, `.` does not match newline. Thus, `spam.*ham` will not match the text if `spam` is on one line and `ham` is on a subsequent line. The `re.SINGLELINE` flag allows `.` to match newline, enabling searches that span lines.

`re.S` is an abbreviation for `re.SINGLELINE`.

Treating text like multiple lines

Normally, `^` only matches the beginning of a string and `$` only matches the end. If you use the `re.MULTILINE` flag, these anchors will also match the beginning and end of embedded lines.

`re.M` is an abbrevation for `re.MULTILINE`.

Example

regex_newlines.py

```python
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim 
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex  
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur. 
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui 
officia deserunt Y-45 mollit anim id est laborum"""

line_start_word = r'^\w+'  # match word at beginning of string/line

matches = re.findall(line_start_word, s)  # only matches at beginning of string
print("matches:", matches)
print()

matches = re.findall(line_start_word, s, re.M)  # matches at beginning of lines
print("matches:", matches)
print()

phrase = r"aliquip.*commodo"

match = re.search(phrase, s)
if match:
    print(match.group(), match.start())
else:
    print(f"{phrase} not found")
print()

match = re.search(phrase, s, re.S)
if match:
    print(repr(match.group()), match.start())
else:
    print(f"{phrase} not found")
print()
```

regex_newlines.py

```
matches: ['lorem']

matches: ['lorem', 'ad', 'ea', 'voluptate', 'Excepteur', 'officia']

aliquip.*commodo not found

'aliquip ex  \nea commodo' 223
```

Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with match.group()

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `:`. This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked with parentheses, and "capture" whatever matched the pattern inside the parentheses.

`re.findall()` returns a list of tuples, where each tuple contains the matches for all each group.

To access groups in more detail, use `re.finditer()` and call the `.group()` method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either `match.group(0)`, or just `match.group()`. Then, `match.group(1)` returns text matched by the first set of parentheses, `match.group(2)` returns the text from the second set, etc.

In the same vein, `match.start()` or `match.start(0)` return the beginning 0-based offset of the entire match; `match.start(1)` returns the beginning offset of group 1, and so forth. The same is true for `match.end()` and `match.end(n)`.

`match.span()` returns the the start and end offsets for the entire match. `match.span(1)` returns start and end offsets for group 1, and so forth.

Example

regex_group.py

```python
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'([A-Z])-(\d{2,3})'  # parens delimit groups

print("Group         start  end")
print("0      1  2")
print("-" * 25)

for m in re.finditer(pattern, s):
    #  (group 0 is entire match)
    print(f"{m.group(0):5s}  {m.group(1)}   {m.group(2):>3s}  {m.start(1):>3d}  {m.end(1
):>3d}")
print()

matches = re.findall(pattern, s)  # findall() returns list of tuples containing groups
print("matches:", matches)
```

regex_group.py

```
Group         start  end
0      1  2
-------------------------
M-302  M   302   12   13
H-476  H   476  102  103
Q-51   Q    51  134  135
A-110  A   110  398  399
H-332  H   332  436  437
Y-45   Y    45  470  471

matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('A', '110'), ('H', '332'), ('Y',
'45')]
```

Special groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark.

Non-capture groups

The most basic special is `(?:pattern)`, which groups but does not capture.

Named groups

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax `` (?`P<name>pattern) ``. You can then call `match.group("name")` to fetch the text match by that sub-expression; alternatively, you can call `match.groupdict()`, which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

Lookaround assertions

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax `(?=pattern)`. The string being matched must match the lookahead, but does not become part of the overall match.

For instance, `\d(?st|nd|rd|th)(?=street)` matches "1st", "2nd", etc., but only where they are followed by "street".

Example

regex_special.py

```python
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])-(?P<number>\d{2,3})'  # Use (?P<NAME>...) to name groups

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number'))  # Use m.group(NAME) to retrieve text
```

regex_special.py

```
M 302
H 476
Q 51
A 110
H 332
Y 45
```

Replacing text

- Use `re.sub(pattern, replacement,string[,count])
- `re.subn(…) returns string and count

To find and replace text using a regular expression, use the `re.sub()` method. It takes the pattern, the replacement text and the string to search as arguments, and returns the modified string.

The third (optional) argument is one or more compilation flags. The fourth argument is the maximum number of replacements to make. Both are optional, but if the fourth argument is specified and no flags are needed, use `0` as a placeholder for the third argument.

TIP Be sure to put the arguments in the proper order!

Example

regex_sub.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) # replace pattern with string
print(s2)
print()

s3, count = rx_code.subn("___", s) # subn returns tuple with result string and
replacement count
print("Made {} replacements".format(count))
print(s3)
```

regex_sub.py

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed do
 eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua. Ut
enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [REDACTED]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa qui
officia deserunt [REDACTED] mollit anim id est laborum

Made 8 replacements
lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do
 eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo ___  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
officia deserunt ___ mollit anim id est laborum
```

Replacing with backrefs

- Use match or groups in replacement text
  - `\g<num>` *group number*
  - `\g<name>` *group name*
  - `\gnum` *shortcut for group number*

It is common to need all or part of the match in the replacement text. To allow this, the `re` module provides *backrefs*, which are special variables that *refer back* to the groups in the match, including group 0 (the entire match).

To refer to a particular group, use the syntax `\g<num>`.

To refer to a particular named group, use `\g<name>`.

As a shortcut, you can use `\num`. This does not work for group 0 — use `\g<0>` instead.

WARNING `\10` is group 10, not group 1 followed by '0'

Example

regex_sub_backrefs.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub(r"(\g<1>)[\g<2>]", s)
print(f"s2: {s2}")
print('-' * 60)

s3 = rx_code.sub(r"\g<number>-\g<letter>", s)
print(f"s3: {s3}")
print('-' * 60)

s4 = rx_code.sub(r"[\1:\2]", s)
print(f"s4: {s4}")
print('-' * 60)
```

regex_sub_backrefs.py

```
s2: lorem ipsum (M)[302] dolor sit amet, consectetur (r)[99] adipiscing elit, sed do
 eiusmod tempor incididunt (H)[476] ut labore et dolore magna (Q)[51] aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo (z)[883]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat (A)[110] cupidatat non proident, sunt in (H)[332] culpa qui
officia deserunt (Y)[45] mollit anim id est laborum
------------------------------------------------------------
s3: lorem ipsum 302-M dolor sit amet, consectetur 99-r adipiscing elit, sed do
 eiusmod tempor incididunt 476-H ut labore et dolore magna 51-Q aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo 883-z  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat 110-A cupidatat non proident, sunt in 332-H culpa qui
officia deserunt 45-Y mollit anim id est laborum
------------------------------------------------------------
s4: lorem ipsum [M:302] dolor sit amet, consectetur [r:99] adipiscing elit, sed do
 eiusmod tempor incididunt [H:476] ut labore et dolore magna [Q:51] aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo [z:883]  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat [A:110] cupidatat non proident, sunt in [H:332] culpa qui
officia deserunt [Y:45] mollit anim id est laborum
------------------------------------------------------------
```

Replacing with a callback

- Replacement can be a callback function
- Function expects match object, returns replacement text
- Use either normally defined function or a lambda

In addition to using a string, possibly containing backrefs, as the replacement, you can specify a function. This function will be called once for each match, with the match object as its only parameter.

Using a callback is necessary if you need to modify any of the original text.

Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to:

- add text around the replacement (can also be done with backrefs)
- search ignoring case and preserve case in a replacement
- look up the text in a dictionary or database to find replacement text

Example

regex_sub_callback.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

def update_code(m):  # callback function is passed each match object
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return '{}:{:04d}'.format(letter, number)  # function returns replacement text


s2, count = rx_code.subn(update_code, s)  # sub takes callback function instead of
replacement text
print(s2)
print(count, "replacements made")
```

regex_sub_callback.py

```
lorem ipsum M:0302 dolor sit amet, consectetur R:0099 adipiscing elit, sed do
 eiusmod tempor incididunt H:0476 ut labore et dolore magna Q:0051 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z:0883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A:0110 cupidatat non proident, sunt in H:0332 culpa qui
officia deserunt Y:0045 mollit anim id est dlaborum
8 replacements made
```

Splitting a string

- Syntax: `re.split(pattern, string[,max])`

The `re.split()` method splits a string into pieces, using the regex to match the delimiters, and returning the pieces as a list. The optional `max` argument limits the numbers of pieces.

Example

regex_split.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
 eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883  consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""


# pattern is one or more non-letters
rx_wordsep = re.compile(r"[^a-z0-9-]+", re.I)  # When splitting, pattern matches what you
don't want

words = rx_wordsep.split(s)  # Retrieve text _separated_ by your pattern
unique_words = set(words)

print(sorted(unique_words))
```

regex_split.py

```
['A-110', 'Duis', 'Excepteur', 'H-332', 'H-476', 'M-302', 'Q-51', 'U901', 'Ut', 'Y-45',
'ad', 'adipiscing', 'aliqua', 'aliquip', 'amet', 'anim', 'aute', 'cillum', 'commodo',
'consectetur', 'consequat', 'culpa', 'cupidatat', 'deserunt', 'dlaborum', 'do', 'dolor',
'dolore', 'ea', 'eiusmod', 'elit', 'enim', 'esse', 'est', 'et', 'eu', 'ex',
'exercitation', 'fugiat', 'id', 'in', 'incididunt', 'ipsum', 'irure', 'labore',
'laboris', 'lorem', 'magna', 'minim', 'mollit', 'nisi', 'non', 'nostrud', 'nulla',
'occaecat', 'officia', 'pariatur', 'proident', 'qui', 'quis', 'r-99', 'reprehenderit',
'sed', 'sint', 'sit', 'sunt', 'tempor', 'ullamco', 'ut', 'velit', 'veniam', 'voluptate',
'z-883']
```

Chapter 1 Exercises

Exercise 1-1 (pyfind.py)

Write a script which takes two or more arguments. The first argument is the pattern to search for; the remaining arguments are files to search. For each file, print out all lines which match the pattern. [1]

Example

```
python pyfind.py freezer DATA/words.txt DATA/parrot.txt
```

Exercise 1-2 (mark_big_words_callback.py, mark_big_words_backrefs.py)

Copy `parrot.txt` to `bigwords.txt` adding asterisks around all words that are 8 or more characters long.

HINT: Use the `\b` anchor to indicate beginning or end of a word.

NOTE There are two solutions to this exercise in the ANSWERS folder: one using a callback function, and one using backrefs.

Exercise 1-3 (print_numbers.py)

Write a script to print out all lines in `custinfo.dat` which contain phone numbers. A phone number consists of three digits, a dash, and four more digits.

Exercise 1-4 (word_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression `[^\w']+` for splitting each line into words.

Test with any of the text files in the DATA folder.

HINT: Use a dictionary for counting.

NOTE The specified pattern matches one or more characters that are neither letters, digits, underscores, nor apostrophes.

[1] Any similarity to the Unix `grep` command is purely intentional.

Index