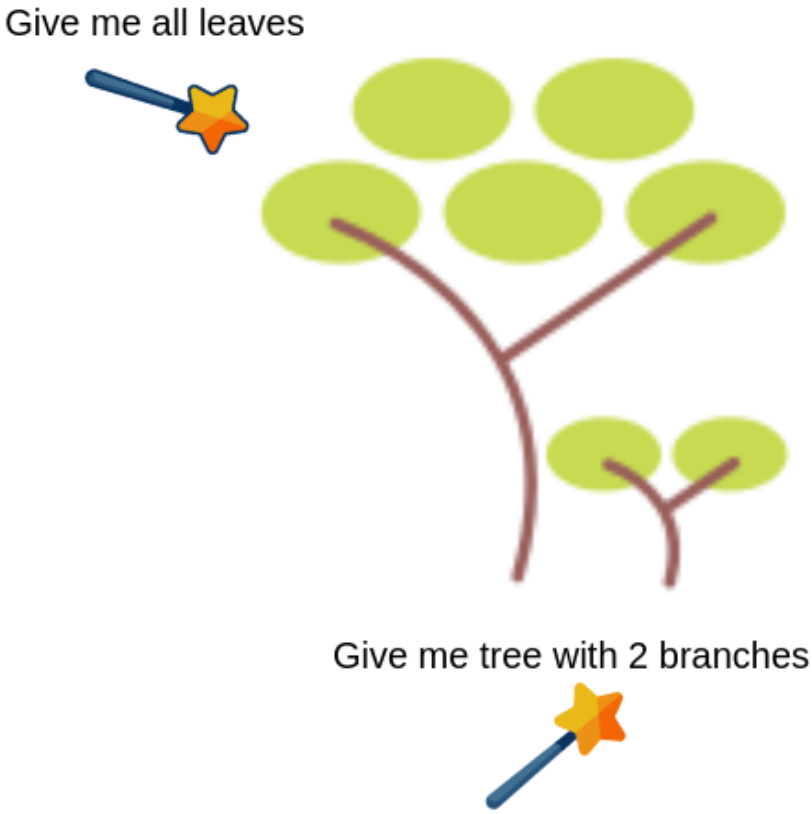

Python re(gex)?

a magical tool for text processing

Sundeep Agarwal

Preface

Scripting and automation tasks often need to extract particular portions of text from input data or modify them from one format to another. This book will help you learn Regular Expressions, a mini-programming language for all sorts of text processing needs.

The book heavily leans on examples to present features of regular expressions one by one. It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to ride a bike or a car - no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

The examples presented here have been tested with **Python version 3.7.1** and may include features not available in earlier versions. Unless otherwise noted, all examples and explanations are meant for ASCII characters only. The examples are copy pasted from Python REPL shell, but modified slightly for presentation purposes (like adding comments and blank lines, shortened error messages, skipping import statements, etc).

Prerequisites

Prior experience working with Python, should know concepts like string formats, string methods, list comprehension and so on.

If you have prior experience with a programming language, but new to Python, check out my GitHub repository on Python Basics before starting this book.

Acknowledgements

- Python documentation - manuals and tutorials
- /r/learnpython/ - helpful forum for beginners and experienced programmers alike
- stackoverflow - for getting answers to pertinent questions on Python and regular expressions
- tex.stackexchange - for help on `pandoc` and `tex` related questions
- draw.io - cover image
- softwareengineering.stackexchange and skolakoda for programming quotes

Special thanks to Al Sweigart, for introducing me to Python with his awesome automatetheboringstuff book and video course.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/py_regular_expressions/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample.

Check out his book on **Ruby Regexp**: https://leanpub.com/rubyregexp

Support the author by donating on patreon or liberapay.

Book version

1.1

Why is it needed?

Regular Expressions have become a synonym with text processing. Most programming languages that are used for scripting purposes come with regular expression module as part of their standard library offering. If not, you can usually find a third-party library support. Syntax and features of regular expressions varies from language to language. Python's offering is similar to that of Perl language, but there are significant differences.

The `str` class comes loaded with variety of methods to deal with text. So, what's so special about regular expressions and why would you need it? For learning and understanding purposes, one can view regular expressions as a mini programming language in itself, specialized for text processing. Parts of a regular expression can be saved for future use, analogous to variables and functions. There are ways to perform AND, OR, NOT conditionals. Operations similar to range function, string repetition operator and so on.

Here's some common use cases:

- sanitizing a string to ensure it satisfies a known set of rules
- filtering or extracting portions on an abstract level like alphabets, numbers, punctuation, etc instead of a known fixed string
- qualified string replacement - start or end of string, whole words, based on surrounding text, etc

Though the term indicates regular structure, modern regular expressions support features like recursion too. So, usage of the term is different than the mathematical concept.

Further Reading

- The true power of regular expressions - it also includes a nice explanation of what regular means
- softwareengineering: Is it a must for every programmer to learn regular expressions?
- softwareengineering: When you should NOT use Regular Expressions?
- Regular Expressions: Now You Have Two Problems
- wikipedia: Regular expression - this article includes discussion on regular expressions as a formal language as well as details on various implementations

Regular Expression modules

In this chapter, you'll get an introduction to two regular expression modules. For some examples, the equivalent normal string method is shown for comparison. Regular expression features will be covered next chapter onwards.

re module

It is always a good idea to know where to find the documentation. The default offering for Python regular expressions is the `re` standard library module. Visit docs.python: re for information on available methods, syntax, features, examples and more. Here's a quote:

> A regular expression (or RE) specifies a set of strings that matches it; the functions in this module let you check if a particular string matches a given regular expression

First up, a simple example to test whether a string is part of another string or not. Normally, you'd use the `in` operator. For regular expressions, use the `re.search` function. Pass the RE as first argument and string to test against as second argument. As a good practice, always use **raw strings** to construct RE, unless other formats are required (will become clearer in coming chapters).

```
>>> sentence = 'This is a sample string'

# check if 'sentence' contains the given string argument
>>> 'is' in sentence
True
>>> 'xyz' in sentence
False

# need to load the re module before use
>>> import re

# check if 'sentence' contains the pattern described by RE argument
>>> bool(re.search(r'is', sentence))
True
>>> bool(re.search(r'xyz', sentence))
False
```

Before using the `re` module, you need to `import` it. Further example snippets will assume that the module is already loaded. The return value of `re.search` function is `re.Match` object when a match is found and `None` otherwise (will be discussed further in a later chapter). For presentation purposes, the examples will use `bool` function to show `True` or `False` depending on whether the RE pattern matched or not.

`bool` is not needed for conditional expressions, output of `re.search` can be used directly.

```
>>> if re.search(r'ring', sentence):
...     print('mission success')
...
mission success

>>> if not re.search(r'xyz', sentence):
...     print('mission failed')
...
mission failed
```

Here's some more examples:

```
>>> words = ['cat', 'attempt', 'tattle']

>>> [w for w in words if re.search(r'tt', w)]
['attempt', 'tattle']
>>> all(re.search(r'at', w) for w in words)
True
>>> any(re.search(r'stat', w) for w in words)
False
```

Compiling regular expressions

Regular expressions can be compiled using `re.compile` function, which gives back a `re.Pattern` object. The top level `re` module functions are all available as methods for this object. Compiling a regular expression helps if the RE has to be used in multiple places or called upon multiple times inside a loop (speed benefit). By default, Python maintains a small list of recently used RE, so the speed benefit doesn't apply for trivial use cases.

```
>>> pet = re.compile(r'dog')
>>> type(pet)
<class 're.Pattern'>

>>> bool(pet.search('They bought a dog'))
True
>>> bool(pet.search('A cat crossed their path'))
False
```

The methods available for compiled patterns might also include some more features than those available for top level functions of `re` module. For example, the `search` method on a compiled pattern has two optional arguments to specify **start** and **end** index. Similar to `range` function and slicing notation, the ending index has to be specified `1` greater than desired index.

```
>>> sentence = 'This is a sample string'
>>> word = re.compile(r'is')

>>> bool(word.search(sentence, 4))
True
>>> bool(word.search(sentence, 6))
False
>>> bool(word.search(sentence, 2, 4))
True
```

bytes

To work with `bytes` data type, the RE must be of `bytes` data as well. Similar to `str` RE, use **raw** form to construct `bytes` RE.

```
>>> sentence = b'This is a sample string'

>>> bool(re.search(rb'is', sentence))
True
>>> bool(re.search(rb'xyz', sentence))
False
```

regex module

The third party `regex` module (https://pypi.org/project/regex/) is backward compatible with standard `re` module as well as offers additional features. This module is lot closer to Perl regular expression in terms of features than the `re` module.

To install the module from command line, you can either use `pip install regex` in a virtual environment or use `python3.7 -m pip install --user regex` for system wide accessibility.

```
>>> import regex
>>> sentence = 'This is a sample string'

>>> bool(regex.search(r'is', sentence))
True
>>> bool(regex.search(r'xyz', sentence))
False
```

You might wonder why two regular expression modules are being presented in this book. The `re` module is good enough for most usecases. But if text processing occupies a large share of your work, the extra features of `regex`

module would certainly come in handy. It would also make it easier to adapt from/to other programming languages. You can also consider always using `regex` module for your project instead of having to decide which one to use depending on features required.

Exercises

Refer to exercises folder for input files required to solve the exercises.

a) For the given input file, print all lines containing the string `two`

```
# note that expected output shown here is wrapped to fit pdf width
>>> filename = 'programming_quotes.txt'
>>> word = re.compile()     ##### add your solution here
>>> with open(filename, 'r') as ip_file:
...     for ip_line in ip_file:
...         if word.search(ip_line):
...             print(ip_line, end='')
...
"Some people, when confronted with a problem, think - I know, I'll use regular expressions.
Now they have two problems" by Jamie Zawinski
"So much complexity in software comes from trying to make one thing do two things" by Ryan Singer
```

b) For the given input string, print all lines NOT containing the string `2`

```
>>> purchases = '''\
... apple 24
... mango 50
... guava 42
... onion 31
... water 10'''
>>> num = re.compile()      ##### add your solution here
>>> for line in purchases.split('\n'):
...     if not num.search(line):
...         print(line)
...
mango 50
onion 31
water 10
```

Anchors

In this chapter, you'll be learning about qualifying a pattern. Instead of matching anywhere in the given input string, restrictions can be specified. For now, you'll see the ones that are already part of `re` module. In later chapters, you'll get to know how to define your own rules for restriction.

These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regular expressions parlance. In case you need to match those characters literally, you need to escape them with a `\` (discussed in a later chapter).

String anchors

This restriction is about qualifying a RE to match only at start or end of an input string. These provide functionality similar to the `str` methods `startswith` and `endswith`. First up, `\A` which restricts the match to start of string.

```
# \A is placed as a prefix to the pattern
>>> bool(re.search(r'\Acat', 'cater'))
True
>>> bool(re.search(r'\Acat', 'concatenation'))
False

>>> bool(re.search(r'\Ahi', 'hi hello\ntop spot'))
True
>>> bool(re.search(r'\Atop', 'hi hello\ntop spot'))
False
```

To restrict the match to end of string, `\Z` is used.

```
# \Z is placed as a suffix to the pattern
>>> bool(re.search(r'are\Z', 'spare'))
True
>>> bool(re.search(r'are\Z', 'nearest'))
False

>>> words = ['surrender', 'unicorn', 'newer', 'door', 'empty', 'eel', 'pest']
>>> [w for w in words if re.search(r'er\Z', w)]
['surrender', 'newer']
>>> [w for w in words if re.search(r't\Z', w)]
['pest']
```

Combining start and end of string anchors, you can restrict the matching to whole string. Similar to comparing strings using the `==` operator.

```
>>> pat = re.compile(r'\Acat\Z')

>>> bool(pat.search('cat'))
True
>>> bool(pat.search('cater'))
False
>>> bool(pat.search('concatenation'))
False
```

Use the optional start/end index arguments for `search` method with caution. They are not equivalent to string slicing. For example, specifying a greater than `0` start index for a RE with start of string anchor is always going to return `False`.

```
>>> pat = re.compile(r'\Aat\Z')

>>> bool(pat.search('cat', 1))
False
>>> bool(pat.search('cat'[1:]))
True
```

The anchors can be used by themselves as a pattern. Helps to insert text at start or end of string, emulating string concatenation operations. These might not feel like useful capability, but combined with other regular expression features they become quite a handy tool. For this illustration, `re.sub` function is used, which performs search and replace operation similar to the normal `replace` string method.

```
# first argument is search RE
# second argument is replace RE
# third argument is string to be acted upon
>>> re.sub(r'\A', r're', 'live')
'relive'
>>> re.sub(r'\A', r're', 'send')
'resend'

>>> re.sub(r'\Z', r'er', 'cat')
'cater'
>>> re.sub(r'\Z', r'er', 'hack')
'hacker'
```

The meaning of RE is widely different when used as a search argument vs replace argument. It will be discussed separately in a later chapter, for now only normal strings will be used as replacement. A common mistake, not specific to `re.sub` , is forgetting that strings are immutable in Python.

```
>>> word = 'cater'
# this will return a string object, won't modify 'word' variable
>>> re.sub(r'\Acat', r'hack', word)
'hacker'
>>> word
'cater'

# need to explicitly assign the result if 'word' has to be changed
>>> word = re.sub(r'\Acat', r'hack', word)
>>> word
'hacker'
```

Line anchors

A string input may contain single or multiple lines. The line separator is the newline character `\n` . So, if you are dealing with Windows OS based text files, you'll have to convert `\r\n` line endings to `\n` first. Which is made easier by Python in many cases - for ex: you can specify which line ending to use for `open` function, the `split` string method handles all whitespaces by default and so on. Or, you can handle `\r` as optional character with quantifiers (covered later).

There are two line anchors, one for start of line and the other for end of line. The `^` metacharacter restricts the matching to start of line, use `$` for end of line. If there are no newline characters in the input string, these will behave same as `\A` and `\Z` respectively.

```
>>> pets = 'cat and dog'

>>> bool(re.search(r'^cat', pets))
True
>>> bool(re.search(r'^dog', pets))
False

>>> bool(re.search(r'dog$', pets))
True

>>> bool(re.search(r'^dog$', pets))
False
```

By default, input string is considered as single line, even if multiple newline characters are present. In such cases, the `$` metacharacter can match both end of string and just before the last newline character. However, `\Z` will always match end of string, irrespective of what characters are present.

```
>>> greeting = 'hi there\nhave a nice day\n'

>>> bool(re.search(r'day$', greeting))
True
>>> bool(re.search(r'day\n$', greeting))
True

>>> bool(re.search(r'day\Z', greeting))
False
>>> bool(re.search(r'day\n\Z', greeting))
True
```

To indicate that the input string should be treated as multiple lines, you need to use the `re.MULTILINE` flag (or, `re.M` short form). The `flags` optional argument will be covered in more detail later.

```
# check if any line in the string starts with 'tap'
>>> bool(re.search(r'^tap', "hi hello\ntop spot", flags=re.M))
False

# check if any line in the string ends with 'ar'
>>> bool(re.search(r'ar$', "spare\npar\ndare", flags=re.M))
True

# filter all elements having lines ending with 'are'
>>> elements = ['spare\n', 'par\n', 'dare']
>>> [e for e in elements if re.search(r'are$', e, flags=re.M)]
['spare\n', 'dare']

# check if any complete line in the string is 'par'
>>> bool(re.search(r'^par$', "spare\npar\ndare", flags=re.M))
True
```

Just like string anchors, you can use the line anchors by themselves as a pattern.

```
>>> ip_lines = "catapults\nconcatenate\ncat"
>>> print(re.sub(r'^', r'* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat

>>> print(re.sub(r'$', r'.', ip_lines, flags=re.M))
catapults.
concatenate.
cat.
```

Word anchors

The third type of restriction is word anchors. A word character is any alphabet (irrespective of case), digit and the underscore character. You might wonder why there are digits and underscores as well, why not only alphabets? This comes from variable and function naming conventions - typically alphabets, digits and underscores are allowed. So, the definition is more programming oriented than natural language.

The escape sequence `\b` denotes a word boundary. This works for both start of word and end of word anchoring. Start of word means either the character prior to the word is a non-word character or there is no character (start of string). Similarly, end of word means the character after the word is a non-word character or no character (end of string). This implies that you cannot have word boundary without a word character.

```
>>> words = 'par spar apparent spare part'

# replace 'par' irrespective of where it occurs
>>> re.sub(r'par', r'X', words)
'X sX apXent sXe Xt'
```

```
# replace 'par' only at start of word
>>> re.sub(r'\bpar', r'X', words)
'X spar apparent spare Xt'
# replace 'par' only at end of word
>>> re.sub(r'par\b', r'X', words)
'X sX apparent spare part'
# replace 'par' only if it is not part of another word
>>> re.sub(r'\bpar\b', r'X', words)
'X spar apparent spare part'
```

You can get lot more creative with using word boundary as a pattern by itself:

```
# space separated words to double quoted csv
# note the use of 'replace' string method, 'translate' method can also be used
>>> print(re.sub(r'\b', r'"', words).replace(' ', ','))
"par","spar","apparent","spare","part"

>>> re.sub(r'\b', r' ', '-----hello-----')
'----- hello -----'

# make a programming statement more readable
# shown for illustration purpose only, won't work for all cases
>>> re.sub(r'\b', r' ', 'foo_baz=num1+35*42/num2')
' foo_baz = num1 + 35 * 42 / num2 '
# excess space at start/end of string can be stripped off
# later you'll learn how to add a qualifier so that strip is not needed
>>> re.sub(r'\b', r' ', 'foo_baz=num1+35*42/num2').strip()
'foo_baz = num1 + 35 * 42 / num2'
```

The word boundary has an opposite anchor too. `\B` matches wherever `\b` doesn't match. This duality will be seen with some other escape sequences too. Negative logic is handy in many text processing situations. But use it with care, you might end up matching things you didn't intend!

```
>>> words = 'par spar apparent spare part'

# replace 'par' if it is not start of word
>>> re.sub(r'\Bpar', r'X', words)
'par sX apXent sXe part'
# replace 'par' at end of word but not whole word 'par'
>>> re.sub(r'\Bpar\b', r'X', words)
'par sX apparent spare part'
# replace 'par' if it is not end of word
>>> re.sub(r'par\B', r'X', words)
'par spar apXent sXe Xt'
# replace 'par' if it is surrounded by word characters
>>> re.sub(r'\Bpar\B', r'X', words)
'par spar apXent sXe part'
```

Here's some standalone pattern usage to compare and contrast the two word anchors:

```
>>> re.sub(r'\b', r':', 'copper')
':copper:'
>>> re.sub(r'\B', r':', 'copper')
'c:o:p:p:e:r'

>>> re.sub(r'\b', r' ', '-----hello-----')
'----- hello -----'
>>> re.sub(r'\B', r' ', '-----hello-----')
' - - - - -h e l l o- - - - - '
```

In this chapter, you've begun to see building blocks of regular expressions and how they can be used in interesting ways. But at the same time, regular expression is but another tool in the land of text processing. Often, you'd get simpler solution by combining regular expressions with other string methods and comprehensions. Practice, experi-

ence and imagination would help you construct creative solutions. In coming chapters, you'll see more applications of anchors as well as `\G` anchor which is best understood in combination with other regular expression features.

Exercises

a) For the given **url**, count the total number of lines that contain `is` or `the` as whole words. Note that each `line` in the `for` loop will be of `bytes` data type.

```
>>> import urllib.request
>>> scarlet_pimpernel_link = r'https://www.gutenberg.org/cache/epub/60/pg60.txt'
>>> word1 = re.compile()    ##### add your solution here
>>> word2 = re.compile()    ##### add your solution here
>>> count = 0
>>> with urllib.request.urlopen(scarlet_pimpernel_link) as ip_file:
...     for line in ip_file:
...         if word1.search(line) or word2.search(line):
...             count += 1
...
>>> print(count)
3737
```

b) For the given input string, change only whole word `red` to `brown`

```
>>> words = 'bred red spread credible'

>>> re.sub()     ##### add your solution here
'bred brown spread credible'
```

c) For the given input list, filter all elements that contains `42` surrounded by word characters.

```
>>> words = ['hi42bye', 'nice1423', 'bad42', 'cool_42a', 'fake4b']

>>> [w for w in words if re.search()]   ##### add your solution here
['hi42bye', 'nice1423', 'cool_42a']
```

d) For the given input list, filter all elements that start with `den` or end with `ly`

```
>>> foo = ['lovely', '1 dentist', '2 lonely', 'eden', 'fly away', 'dent']

>>> [e for e in foo if ]        ##### add your solution here
['lovely', '2 lonely', 'dent']
```

e) For the given input string, change whole word `mall` only if it is at start of line.

```
>>> para = '''\
... ball fall wall tall
... mall call ball pall
... wall mall ball fall'''

>>> print(re.sub())    ##### add your solution here
ball fall wall tall
1234 call ball pall
wall mall ball fall
```

Alternation and Grouping

Many a times, you'd want to search for multiple terms. In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The RE will match if any of the expression separated by `|` is satisfied. These can have their own independent anchors as well.

```
# match either 'cat' or 'dog'
>>> bool(re.search(r'cat|dog', 'I like cats'))
True
>>> bool(re.search(r'cat|dog', 'I like dogs'))
True
>>> bool(re.search(r'cat|dog', 'I like parrots'))
False

# replace either 'cat' at start of string or 'cat' at end of word
>>> re.sub(r'\Acat|cat\b', r'X', 'catapults concatenate cat scat')
'Xapults concatenate X sX'

# replace either 'cat' or 'dog' or 'fox' with 'mammal'
>>> re.sub(r'cat|dog|fox', r'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

You might infer from above examples that there can be cases where lots of alternation is required. The `join` string method can be used to build the alternation list automatically from an iterable of strings.

```
>>> '|'.join(['car', 'jeep'])
'car|jeep'

>>> words = ['cat', 'dog', 'fox']
>>> '|'.join(words)
'cat|dog|fox'

>>> re.sub('|'.join(words), r'mammal', 'cat dog bee parrot fox')
'mammal mammal bee parrot mammal'
```

Often, there are some common things among the RE alternatives. It could be common characters or qualifiers like the anchors. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c) = ab+ac` in maths, you get `a(b|c) = ab|ac` in RE.

```
# without grouping
>>> re.sub(r'reform|rest', r'X', 'red reform read arrest')
'red X read arX'
# with grouping
>>> re.sub(r're(form|st)', r'X', 'red reform read arrest')
'red X read arX'

# without grouping
>>> re.sub(r'\bpar\b|\bpart\b', r'X', 'par spare part party')
'X spare X party'
# taking out common anchors
>>> re.sub(r'\b(par|part)\b', r'X', 'par spare part party')
'X spare X party'
# taking out common characters as well
# you'll later learn a better technique instead of using empty alternate
>>> re.sub(r'\bpar(|t)\b', r'X', 'par spare part party')
'X spare X party'
```

There's lot more features to grouping than just forming terser RE. For now, this is a good place to show how to incorporate normal strings (could be a variable, result from an expression, etc) while building a regular expression. For ex: adding anchors to alternation list created using the `join` method.

```
>>> words = ['cat', 'par']
>>> '|'.join(words)
'cat|par'
# without word boundaries, any matching portion will be replaced
>>> re.sub('|'.join(words), r'X', 'cater cat concatenate par spare')
'Xer X conXenate X sXe'

# note how raw string is used on either side of concatenation
# avoid f-strings unless you know how to compensate for RE
>>> alt = re.compile(r'\b(' + '|'.join(words) + r')\b')
# only whole words will be replaced now
>>> alt.sub(r'X', 'cater cat concatenate par spare')
'cater X concatenate X spare'

# this is how the above RE looks as a normal string
>>> alt.pattern
'\\b(cat|par)\\b'
>>> alt.pattern == r'\b(cat|par)\b'
True
```

In the above examples with `join` method, the string iterable elements do not contain any special regular expression characters. How to deal strings with special characters will be discussed in a later chapter.

Precedence rules

There's some tricky situations when using alternation. If it is used for testing a match to get `True/False` against a string input, there is no ambiguity. However, for other things like string replacement, it depends on a few factors. Say, you want to replace either `are` or `spared` - which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

In Python, the alternative which matches earliest in the input string gets precedence.

```
>>> words = 'lion elephant are rope not'

# span shows the start and end+1 index of matched portion
>>> re.search(r'on', words)
<re.Match object; span=(2, 4), match='on'>
>>> re.search(r'ant', words)
<re.Match object; span=(10, 13), match='ant'>

# starting index of 'on' < index of 'ant' for given string input
# so 'on' will be replaced irrespective of order
# count optional argument here restricts no. of replacements to 1
>>> re.sub(r'on|ant', r'X', words, count=1)
'liX elephant are rope not'
>>> re.sub(r'ant|on', r'X', words, count=1)
'liX elephant are rope not'
```

What happens if alternatives match on same index? The precedence is then left to right in the order of declaration.

```
>>> mood = 'best years'
>>> re.search(r'year', mood)
<re.Match object; span=(5, 9), match='year'>
>>> re.search(r'years', mood)
<re.Match object; span=(5, 10), match='years'>

# starting index for 'year' and 'years' will always be same
# so, which one gets replaced depends on the order of alternation
>>> re.sub(r'year|years', r'X', mood, count=1)
'best Xs'
>>> re.sub(r'years|year', r'X', mood, count=1)
'best X'
```

Another example (without `count` restriction) to drive home the issue:

```
>>> words = 'ear xerox at mare part learn eye'

# this is going to be same as: r'ar'
>>> re.sub(r'ar|are|art', r'X', words)
'eX xerox at mXe pXt leXn eye'
# this is going to be same as: r'are|ar'
>>> re.sub(r'are|ar|art', r'X', words)
'eX xerox at mX pXt leXn eye'
# phew, finally this one works as needed
>>> re.sub(r'are|art|ar', r'X', words)
'eX xerox at mX pX leXn eye'
```

If you do not want substrings to sabotage your replacements, a robust workaround is to sort the alternations based on length, longest first.

```
>>> words = ['hand', 'handy', 'handful']
>>> '|'.join(sorted(words, key=len, reverse=True))
'handful|handy|hand'
>>> alt = re.compile('|'.join(sorted(words, key=len, reverse=True)))

>>> alt.sub(r'X', 'hands handful handed handy')
'Xs X Xed X'
# without sorting, alternation order will come into play
>>> re.sub('|'.join(words), r'X', 'hands handful handed handy')
'Xs Xful Xed Xy'
```

So, this chapter was about specifying one or more alternate matches within the same RE using `|` metacharacter. Which can further be simplified using `()` grouping if the alternations have common aspects. Among the alternations, earliest matching pattern gets precedence. Left to right ordering is used as a tie-breaker if multiple alternations match starting from same location. You also learnt ways to programmatically construct a RE.

Exercises

a) For the given input list, filter all elements that start with `den` or end with `ly`

```
>>> foo = ['lovely', '1 dentist', '2 lonely', 'eden', 'fly away', 'dent']

>>> [e for e in foo if ]     ##### add your solution here
['lovely', '2 lonely', 'dent']
```

b) For the given **url**, count the total number of lines that contain `removed` or `rested` or `received` or `replied` or `refused` or `retired` as whole words. Note that each `line` in the `for` loop will be of `bytes` data type.

```
>>> import urllib.request
>>> scarlet_pimpernel_link = r'https://www.gutenberg.org/cache/epub/60/pg60.txt'
>>> words = re.compile()   ##### add your solution here
>>> count = 0
>>> with urllib.request.urlopen(scarlet_pimpernel_link) as ip_file:
...     for line in ip_file:
...         if words.search(line):
...             count += 1
...
>>> print(count)
83
```

Escaping metacharacters

You have seen a few metacharacters and escape sequences that help to compose a RE. To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` character. To indicate a literal `\` character, use `\\` . Assuming these are all part of raw string, not normal strings.

```
# even though ^ is not being used as anchor, it won't be matched literally
>>> bool(re.search(r'b^2', 'a^2 + b^2 - C*3'))
False
# escaping will work
>>> bool(re.search(r'b\^2', 'a^2 + b^2 - C*3'))
True

# match ( or ) literally
>>> re.sub(r'\(|\)', r'', '(a*b) + c')
'a*b + c'

# note that here input string is also a raw string
>>> re.sub(r'\\', r'/', r'\learn\by\example')
'/learn/by/example'
```

As emphasized earlier, regular expression is just another tool to process text. Some examples and exercises presented in this book can be solved using normal string methods as well. For real world use cases, ask yourself first if regular expression is needed at all?

```
>>> eqn = 'f*(a^b) - 3*(a^b)'

# straightforward search and replace, no need RE shenanigans
>>> eqn.replace('(a^b)', 'c')
'f*c - 3*c'
```

Okay, what if you have a string variable that must be used to construct a RE - how to escape all the metacharacters? Relax, `re.escape` function has got you covered. No need to manually take care of all the metacharacters or worry about changes in future versions.

```
>>> expr = '(a^b)'
# print used here to show results similar to raw string
>>> print(re.escape(expr))
\(a\^b\)

# if strings are to be matched literally,
# need to use re.escape for each string when creating alternations
>>> terms = ['foo_baz', expr]
>>> print('|'.join(re.escape(w) for w in terms))
foo_baz|\(a\^b\)

# replace only at end of string
>>> re.sub(re.escape(expr) + r'\Z', r'c', eqn)
'f*(a^b) - 3*c'
```

Exercises

a) Transform given input strings to expected output using same logic on both strings.

```
>>> str1 = '(9-2)*5+qty/3'
>>> str2 = '(qty+4)/2-(9-2)*5+pq/4'

>>>    ##### add your solution here for str1
'35+qty/3'
>>>    ##### add your solution here for str2
'(qty+4)/2-35+pq/4'
```

b) Replace any matching item from given list with `X` for given input strings.

```
>>> items = ['a.b', '3+n', r'x\y\z', 'qty||price', '{n}']
>>> alt_re = re.compile()      ##### add your solution here

>>> alt_re.sub(r'X', '0a.bcd')
'0Xcd'
>>> alt_re.sub(r'X', 'E{n}AMPLE')
'EXAMPLE'
>>> alt_re.sub(r'X', r'43+n2 ax\y\ze')
'4X2 aXe'
```

Dot metacharacter and Quantifiers

As an analogy, alternation provides logical OR. Combining the dot metacharacter `.` and quantifiers (and alternation if needed) paves a way to perform logical AND. For example, you want to check if a string matches two patterns with any number of characters in between. The dot metacharacter serves as a placeholder to match any character except the newline character. In later chapters, you'll learn how to include the newline character, as well as how to define your own custom placeholder for limited set of characters.

```
# matches character 'c', any character and then character 't'
>>> re.sub(r'c.t', r'X', 'tac tin cat abc;tuv acute')
'taXin X abXuv aXe'

# matches character 'r', any two characters and then character 'd'
>>> re.sub(r'r..d', r'X', 'breadth markedly reported overrides')
'bXth maXly repoX oveXes'

# matches character '2', any character and then character '3'
>>> re.sub(r'2.3', r'8', '42\t33')
'483'
```

Greedy quantifiers

Quantifiers are like string repetition operator and range function. They can be applied to both characters and groupings. Apart from ability to specify exact quantity and bounded range, these can also match unbounded varying quantities. If the input string can satisfy a pattern with varying quantities in multiple ways, you can choose among three types of quantifiers to narrow down possibilities. In this section, greedy type of quantifiers is covered.

First up, the `?` metacharacter which quantifies a character or group to match `0` or `1` times. This helps to define optional patterns and build terser RE compared to groupings for some cases.

```
# same as: r'ear|ar'
>>> re.sub(r'e?ar', r'X', 'far feat flare fear')
'fX feat flXe fX'

# same as: r'\bpar(t|)\b'
>>> re.sub(r'\bpart?\b', r'X', 'par spare part party')
'X spare X party'

# same as: r'\b(re.d|red)\b'
>>> words = ['red', 'read', 'ready', 're;d', 'redo', 'reed']
>>> [w for w in words if re.search(r'\bre.?d\b', w)]
['red', 'read', 're;d', 'reed']

# same as: r'part|parrot'
>>> re.sub(r'par(ro)?t', r'X', 'par part parrot parent')
'par X X parent'
# same as: r'part|parrot|parent'
>>> re.sub(r'par(en|ro)?t', r'X', 'par part parrot parent')
'par X X X'
```

The `*` metacharacter quantifies a character or group to match `0` or more times. There is no upper bound, more details will be discussed at end of this section.

```
# match 't' followed by zero or more of 'a' followed by 'r'
>>> re.sub(r'ta*r', r'X', 'tr tear tare steer sitaara')
'X tear Xe steer siXa'
# match 't' followed by zero or more of 'e' or 'a' followed by 'r'
>>> re.sub(r't(e|a)*r', r'X', 'tr tear tare steer sitaara')
'X X Xe sX siXa'
# match zero or more of '1' followed by '2'
>>> re.sub(r'1*2', r'X', '3111111111125111142')
'3X511114X'
```

Time to introduce more `re.split` function:

```
# last element is empty because there is nothing between 511114 and 2
>>> re.split(r'1*2', '3111111111125111142')
['3', '511114', '']

# optional argument maxsplit specifies how many times to split
# later, you'll see how to get behavior like the str.partition method
>>> re.split(r'1*2', '3111111111125111142', maxsplit=1)
['3', '5111142']

# empty string matches at start and end of string
# it matches between every character
# and, there is an empty match after the split at u
>>> re.split(r'u*', 'cloudy')
['', 'c', 'l', 'o', '', 'd', 'y', '']
```

The `+` metacharacter quantifies a character or group to match `1` or more times. Similar to `*` quantifier, there is no upper bound. More importantly, this doesn't have surprises like matching empty string in between patterns or at start/end of string.

```
>>> re.sub(r'ta+r', r'X', 'tr tear tare steer sitaara')
'tr tear Xe steer siXa'
>>> re.sub(r't(e|a)+r', r'X', 'tr tear tare steer sitaara')
'tr X Xe sX siXa'

>>> re.sub(r'1+2', r'X', '3111111111125111142')
'3X5111142'
>>> re.split(r'1+', '3111111111125111142')
['3', '25', '42']
>>> re.split(r'u+', 'cloudy')
['clo', 'dy']
```

You can specify a range of integer numbers, both bounded and unbounded, using `{}` metacharacters. There are four ways to use this quantifier as listed below:

- `{m,n}` will match `m` to `n` times
- `{m,}` will match at least `m` times
- `{,n}` will match up to `n` times (including `0` times)
- `{n}` will match exactly `n` times

```
>>> demo = ['abc', 'ac', 'adc', 'abbc', 'xabbbcz', 'bbb', 'bc', 'abbbbbc']

>>> [w for w in demo if re.search(r'ab{1,4}c', w)]
['abc', 'abbc', 'xabbbcz']
>>> [w for w in demo if re.search(r'ab{3,}c', w)]
['xabbbcz', 'abbbbbc']
>>> [w for w in demo if re.search(r'ab{,2}c', w)]
['abc', 'ac', 'abbc']
>>> [w for w in demo if re.search(r'ab{3}c', w)]
['xabbbcz']
```

Note: The `{}` metacharacters have to be escaped to match them literally. However, unlike `()` metacharacters, these have lot more leeway. For ex: escaping `{` alone is enough, or if it doesn't conform strictly to any of the four forms listed above, escaping is not needed at all.

Next up, how to construct AND conditional using dot metacharacter and quantifiers.

```
# match 'Error' followed by zero or more characters followed by 'valid'
>>> bool(re.search(r'Error.*valid', 'Error: not a valid input'))
True

>>> bool(re.search(r'Error.*valid', 'Error: key not found'))
False
```

To allow matching in any order, you'll have to bring in alternation as well. That is somewhat manageable for 2 or 3 patterns. In a later chapter, you'll learn how to use lookarounds for a comparatively easier approach.

```
>>> seq1 = 'cat and dog'
>>> seq2 = 'dog and cat'
>>> bool(re.search(r'cat.*dog|dog.*cat', seq1))
True
>>> bool(re.search(r'cat.*dog|dog.*cat', seq2))
True

# if you just need True/False result, this would be a scalable approach
>>> patterns = (r'cat', r'dog')
>>> all(re.search(p, seq1) for p in patterns)
True
>>> all(re.search(p, seq2) for p in patterns)
True
```

So, how much do these greedy quantifiers match? When you are using `?` how does Python decide to match `0` or `1` times, if both quantities can satisfy the RE? For example, consider the expression `re.sub(r'f.?o', r'X', 'foot')` - should `foo` be replaced or `fo` ? It will always replace `foo` , because these are **greedy** quantifiers, meaning longest match wins.

```
>>> re.sub(r'f.?o', r'X', 'foot')
'Xt'

# a more practical example
# prefix '<' with '\' if it is not already prefixed
>>> print(re.sub(r'\\?<', r'\<', r'blah \< foo < bar \< blah < baz'))
blah \< foo \< bar \< blah \< baz

# say goodbye to /handful|handy|hand/ shenanigans
>>> re.sub(r'hand(y|ful)?', r'X', 'hand handy handful')
'X X X'
```

But wait, how did `r'Error.*valid'` example work? Shouldn't `.*` consume all the characters after `Error` ? Good question. The regular expression engine actually does consume all the characters. Then realizing that the RE fails, it gives back one character from end of string and checks again if RE is satisfied. This process is repeated until a match is found or failure is confirmed. In regular expression parlance, this is called **backtracking**. And can be quite time consuming for certain corner cases.

```
>>> sentence = 'that is quite a fabricated tale'

# r't.*a' will always match from first 't' to last 'a'
# also, note that count argument is set to 1 for illustration purposes
>>> re.sub(r't.*a', r'X', sentence, count=1)
'Xle'
>>> re.sub(r't.*a', r'X', 'star', count=1)
'sXr'

# matching first 't' to last 'a' for t.*a won't work for these cases
# the engine backtracks until .*q matches and so on
>>> re.sub(r't.*a.*q.*f', r'X', sentence, count=1)
'Xabricated tale'
>>> re.sub(r't.*a.*u', r'X', sentence, count=1)
'Xite a fabricated tale'
```

Non-greedy quantifiers

As the name implies, these quantifiers will try to match as minimally as possible. Also known as **lazy** or **reluctant** quantifiers. Appending a `?` to greedy quantifiers makes them non-greedy.

```
>>> re.sub(r'f.??o', r'X', 'foot', count=1)
'Xot'

>>> re.sub(r'f.??o', r'X', 'frost', count=1)
'Xst'

>>> re.sub(r'.{2,5}?', r'X', '123456789', count=1)
'X3456789'
```

Like greedy quantifiers, lazy quantifiers will try to satisfy the overall RE.

```
>>> sentence = 'that is quite a fabricated tale'

# r't.*?a' will always match from first 't' to first 'a'
>>> re.sub(r't.*?a', r'X', sentence, count=1)
'Xt is quite a fabricated tale'

# matching first 't' to first 'a' for t.*?a won't work for this case
# so, engine will move forward until .*?f matches and so on
>>> re.sub(r't.*?a.*?f', r'X', sentence, count=1)
'Xabricated tale'
```

Possessive quantifiers

Note: This feature is not present in `re` module, but is offered by the `regex` module.

Appending a `+` to greedy quantifiers makes them possessive. These are like greedy quantifiers, but without the backtracking. So, something like `r'Error.*+valid'` will never match because `.*+` will consume all the remaining characters. If both greedy and possessive quantifier versions are functionally equivalent, then possessive is preferred because it will fail faster for non-matching cases. In a later chapter, you'll see an example where a RE will only work with possessive quantifier, but not if greedy quantifier is used.

```
>>> import regex
>>> demo = ['abc', 'ac', 'adc', 'abbc', 'xabbbcz', 'bbb', 'bc', 'abbbbbc']

# functionally equivalent greedy and possessive versions
>>> [w for w in demo if regex.search(r'ab*c', w)]
['abc', 'ac', 'abbc', 'xabbbcz', 'abbbbbc']
>>> [w for w in demo if regex.search(r'ab*+c', w)]
['abc', 'ac', 'abbc', 'xabbbcz', 'abbbbbc']

# different results
>>> regex.sub(r'f(a|e)*at', r'X', 'feat ft feaeat')
'X ft X'
# (a|e)*+ would match 'a' or 'e' as much as possible
# no backtracking, so another 'a' can never match
>>> regex.sub(r'f(a|e)*+at', r'X', 'feat ft feaeat')
'feat ft feaeat'
```

The effect of possessive quantifier can also be expressed using **atomic grouping**. The syntax is `(?>RE)` - in later chapters you'll see more such special groupings.

```
# same as: r'(b|o)++'
>>> regex.sub(r'(?>(b|o)+)', r'X', 'abbbc fooooooot')
'aXc fXt'

# same as: r'f(a|e)*+at'
>>> regex.sub(r'f(?>(a|e)*)at', r'X', 'feat ft feaeat')
'feat ft feaeat'
```

This chapter introduced the concept of specifying a placeholder instead of fixed string. Combined with quantifiers, you've seen a glimpse of how a simple RE can match wide range of text. In coming chapters, you'll learn how to create your own restricted set of placeholder characters.

Exercises

Note that some exercises are intentionally designed to be complicated to solve with regular expressions alone. Try to use normal string methods, break down the problem into multiple steps, etc. Some exercises will become easier to solve with techniques presented in chapters to come. Going through the exercises a second time after finishing entire book will be fruitful as well.

a) Use regular expression to get the output as shown for the given strings.

```
>>> eqn1 = 'a+42//5-c'
>>> eqn2 = 'pressure*3+42/5-14256'
>>> eqn3 = 'r*42-5/3+42///5-42/53+a'

##### add your solution here for eqn1
['a+', '-c']
##### add your solution here for eqn2
['pressure*3+', '-14256']
##### add your solution here for eqn3
['r*42-5/3+42///5-', '3+a']
```

b) For the given strings, construct a RE to get output as shown.

```
>>> str1 = 'a+b(addition)'
>>> str2 = 'a/b(division) + c%d(#modulo)'
>>> str3 = 'Hi there(greeting). Nice day(a(b)'

>>> remove_parentheses = re.compile()     ##### add your solution here
>>> remove_parentheses.sub('', str1)
'a+b'
>>> remove_parentheses.sub('', str2)
'a/b + c%d'
>>> remove_parentheses.sub('', str3)
'Hi there. Nice day'
```

c) Remove leading/trailing whitespaces from all the individual fields of these csv strings.

```
>>> csv1 = ' comma  ,separated ,values '
>>> csv2 = 'good bad,nice  ice  , 42 , ,   stall   small'

##### add your solution here for csv1
'comma,separated,values'
##### add your solution here for csv2
'good bad,nice  ice,42,,stall   small'
```

d) Correct the given RE to get the expected output.

```
>>> words = 'plink incoming tint winter in caution sentient'
>>> change = re.compile(r'int|in|ion|ing|inco|inter|ink')

# wrong output
>>> change.sub(r'X', words)
'plXk XcomXg tX wXer X cautX sentient'

# expected output
>>> change = re.compile()       ##### add your solution here
>>> change.sub(r'X', words)
'plX XmX tX wX X cautX sentient'
```

e) For the given greedy quantifiers, what would be the equivalent form using `{m,n}` representation?

- `?` is same as
- `*` is same as
- `+` is same as

f) `(a*|b*)` is same as `(a|b)*` - True or False?

Working with matched portions

Having seen a few features that can match varying text, you'll learn how to extract and work with those matching portions in this chapter.

re.Match object

The `re.search` function returns a `re.Match` object from which various details can be extracted like the matched portion of string, location of matched portion, etc. See docs.python: Match Objects for details.

```
>>> re.search(r'ab*c', 'abc ac adc abbbc')
<re.Match object; span=(0, 3), match='abc'>

>>> re.search(r'b.*d', 'abc ac adc abbbc')
<re.Match object; span=(1, 9), match='bc ac ad'>
```

The RE grouping inside `()` is also known as a **capture group**. It has multiple uses, one of which is the ability to work with matched portions of those groups. When capture groups are used with `re.search`, they can be retrieved using an index on the `re.Match` object. The first element is always the entire matched portion and rest of the elements are for capture groups if they are present. The leftmost `(` will get group number `1`, second leftmost `(` will get group number `2` and so on.

```
>>> re.search(r'b.*d', 'abc ac adc abbbc')
<re.Match object; span=(1, 9), match='bc ac ad'>
# retrieving entire matched portion
>>> re.search(r'b.*d', 'abc ac adc abbbc')[0]
'bc ac ad'
# can also pass an index by calling 'group' method Match object
>>> re.search(r'b.*d', 'abc ac adc abbbc').group(0)
'bc ac ad'

# capture group example
>>> m = re.search(r'a(.*)d(.*a)', 'abc ac adc abbbc')
# to get matched portion of second capture group
>>> m[2]
'c a'
# to get a tuple of all the capture groups
>>> m.groups()
('bc ac a', 'c a')
```

Functions can be used in replacement section of `re.sub` instead of a string. A `re.Match` object will be passed to the function as argument. In later chapters, you'll see a way to directly reference the matches in replacement section string.

```
# m[0] will contain entire matched portion
# a^2 and b^2 for the two matches in this example
>>> re.sub(r'(a|b)\^2', lambda m: m[0].upper(), 'a^2 + b^2 - C*3')
'A^2 + B^2 - C*3'
```

re.findall

The `re.findall` function returns all the matched portions as a list.

```
>>> re.findall(r'ab*c', 'abc ac adc abbbc')
['abc', 'ac', 'abbbc']

>>> re.findall(r'ab+c', 'abc ac adc abbbc')
['abc', 'abbbc']

>>> re.findall(r'\bs?pare?\b', 'par spar apparent spare part pare')
['par', 'spar', 'spare', 'pare']
```

It is useful for debugging purposes as well, for example to see what is going on under the hood before applying substitution.

```
>>> re.findall(r't.*a', 'that is quite a fabricated tale')
['that is quite a fabricated ta']

>>> re.findall(r't.*?a', 'that is quite a fabricated tale')
['tha', 't is quite a', 'ted ta']
```

If capture groups are used, each element of output will be a tuple of strings of all the capture groups. Text matched by the RE outside of capture groups won't be present in the output list. If there is only one capture group, tuple won't be used and each element will be the matched portion of that capture group.

```
>>> re.findall(r'a(b*)c', 'abc ac adc abbc xabbbcz bbb bc abbbbbc')
['b', '', 'bb', 'bbb', 'bbbbb']

>>> re.findall(r'(x*):(y*)', 'xx:yyy x: x:yy :y')
[('xx', 'yyy'), ('x', ''), ('x', 'yy'), ('', 'y')]
```

re.finditer

Use `re.finditer` to get a Match object for each matched portion.

```
>>> re.finditer(r'ab+c', 'abc ac adc abbbc')
<callable_iterator object at 0x7fb65e103438>
>>> m_iter = re.finditer(r'ab+c', 'abc ac adc abbbc')
>>> for m in m_iter:
...     print(m)
...
<re.Match object; span=(0, 3), match='abc'>
<re.Match object; span=(11, 16), match='abbbc'>

# same as: re.findall(r'(x*):(y*)', 'xx:yyy x: x:yy :y')
>>> m_iter = re.finditer(r'(x*):(y*)', 'xx:yyy x: x:yy :y')
>>> [(m[1], m[2]) for m in m_iter]
[('xx', 'yyy'), ('x', ''), ('x', 'yy'), ('', 'y')]
```

Here's some more examples:

```
# work with entire matched portions
>>> m_iter = re.finditer(r'ab+c', 'abc ac adc abbbc')
>>> for m in m_iter:
...     print(m[0].upper())
...
ABC
ABBBC

# to get start and end+1 index of entire matched portion
# pass a number as argument to get span of that particular capture group
>>> m_iter = re.finditer(r'ab+c', 'abc ac adc abbbc')
>>> for m in m_iter:
...     print(m.span())
...
(0, 3)
(11, 16)
```

This chapter introduced different ways to work with various matching portions of input string. You learnt another use of groupings and you'll see even more uses of groupings later on.

Exercises

a) For the given strings, extract the matching portion from first `is` to last `t`

```
>>> str1 = 'What is the biggest fruit you have seen?'
>>> str2 = 'Your mission is to read and practice consistently'
>>> expr = re.compile()     ##### add your solution here

>>> expr        ##### add your solution here
'is the biggest fruit'
>>> expr        ##### add your solution here
'ission is to read and practice consistent'
```

b) Transform the given input strings to expected output as shown.

```
>>> row1 = '-2,5 4,+3 +42,-53 '
##### add your solution here
[3, 7, -11]

>>> row2 = '1.32,-3.14 634,5.63 '
##### add your solution here
[-1.82, 639.63]
```

Character class

To create a custom placeholder for limited set of characters, enclose them inside `[]` metacharacters. It is similar to using single character alternations inside a grouping, but without the drawbacks of a capture group. In addition, character classes have their own versions of metacharacters and provide special predefined sets for common use cases. Quantifiers are also applicable to character classes.

```
>>> words = ['cute', 'cat', 'cot', 'coat', 'cost', 'scuttle']
# same as: r'cot|cut' or r'c(o|u)t'
>>> [w for w in words if re.search(r'c[ou]t', w)]
['cute', 'cot', 'scuttle']

# same as: r'(a|e|o)+t'
>>> re.sub(r'[aeo]+t', r'X', 'meeting cute boat site foot')
'mXing cute bX site fX'

>>> re.findall(r'[0123456789]+', 'Sample123string42with777numbers')
['123', '42', '777']
```

Metacharacters

Character classes have their own metacharacters to help define the sets succinctly. Metacharacters outside of character classes like `^`, `$`, `()` etc either don't have special meaning or have completely different one inside the character classes. First up, the `-` metacharacter that helps to define a range of characters instead of having to specify them all individually.

```
# all digits
>>> re.findall(r'[0-9]+', 'Sample123string42with777numbers')
['123', '42', '777']

# whole words made up of lowercase alphabets only
>>> re.findall(r'\b[a-z]+\b', 'coat Bin food tar12 best')
['coat', 'food', 'best']

# whole words made up of lowercase alphabets and digits only
>>> re.findall(r'\b[a-z0-9]+\b', 'coat Bin food tar12 best')
['coat', 'food', 'tar12', 'best']

# whole words made up of lowercase alphabets, but starting with 'p' to 'z'
>>> re.findall(r'\b[p-z][a-z]*\b', 'coat tin food put stoop best')
['tin', 'put', 'stoop']

# whole words made up of only 'a' to 'f' and 'p' to 't' lowercase alphabets
>>> re.findall(r'\b[a-fp-t]+\b', 'coat tin food put stoop best')
['best']
```

Character classes can also be used to construct numeric ranges. However, it is easy to miss corner cases and some ranges are complicated to design.

```
# numbers between 10 to 29
>>> re.findall(r'\b[12][0-9]\b', '23 154 12 26 98234')
['23', '12', '26']

# numbers >= 100
>>> re.findall(r'\b[0-9]{3,}\b', '23 154 12 26 98234')
['154', '98234']

# numbers >= 100 if there are leading zeros
>>> re.findall(r'\b0*[1-9][0-9]{2,}\b', '0501 035 154 12 26 98234')
['0501', '154', '98234']
```

If numeric range is difficult to construct, better to convert the matched portion to appropriate numeric format first.

```
# numbers < 350
>>> m_iter = re.finditer(r'[0-9]+', '45 349 651 593 4 204')
>>> [m[0] for m in m_iter if int(m[0]) < 350]
['45', '349', '4', '204']

# note that return value is string and s[0] is used to get matched portion
>>> def num_range(s):
...     return '1' if 200 <= int(s[0]) <= 650 else '0'
...

# numbers between 200 and 650
# note that only function name is supplied, () is not used
# Match object is automatically passed as argument
>>> re.sub(r'[0-9]+', num_range, '45 349 651 593 4 204')
'0 1 0 1 0 1'
```

Next metacharacter is `^` which has to specified as the first character of the character class. It negates the set of characters, so all characters other than those specified will be matched. As highlighted earlier, handle negative logic with care, you might end up matching more than you wanted. Also, these examples below are all excellent places to use possessive quantifier as there is no backtracking involved.

```
# all non-digits
>>> re.findall(r'[^0-9]+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']

# deleting characters from start of string based on a delimiter
>>> re.sub(r'\A[^=]+', r'', 'foo=42; baz=123', count=1)
'=42; baz=123'
# remove first two columns where : is delimiter
>>> re.sub(r'\A([^:]+:){2}', r'', 'foo:123:bar:baz', count=1)
'bar:baz'

# deleting characters at end of string based on a delimiter
>>> re.sub(r'=[^=]+\Z', r'', 'foo=42; baz=123', count=1)
'foo=42; baz'
```

Sometimes, it is easier to use positive character class and negate the `re.search` result instead of using negated character class.

```
>>> words = ['tryst', 'fun', 'glyph', 'pity', 'why']

# words not containing vowel characters
>>> [w for w in words if re.search(r'\A[^aeiou]+\Z', w)]
['tryst', 'glyph', 'why']

# easier to write and maintain
>>> [w for w in words if not re.search(r'[aeiou]', w)]
['tryst', 'glyph', 'why']
```

Similar to other metacharacters, prefix `\` to character class metacharacters to match them literally. Some of them can be achieved by different placement as well.

```
# - should be first or last character or escaped using \
>>> re.findall(r'\b[a-z-]{2,}\b', 'ab-cd gh-c 12-423')
['ab-cd', 'gh-c']
>>> re.findall(r'\b[a-z\-0-9]{2,}\b', 'ab-cd gh-c 12-423')
['ab-cd', 'gh-c', '12-423']

# ^ should be other than first character or escaped using \
>>> re.findall(r'a[+^]b', 'f*(a^b) - 3*(a+b)')
['a^b', 'a+b']
>>> re.findall(r'a[\^+]b', 'f*(a^b) - 3*(a+b)')
['a^b', 'a+b']
```

```
# [ can be escaped with \ or placed as last character
# ] can be escaped with \ or placed as first character
>>> re.search(r'[a-z\[\]0-9]+', 'words[5] = tea')
<re.Match object; span=(0, 8), match='words[5]'>
# \ should be escaped using \
>>> print(re.search(r'[a\\b]+', r'5ba\babc2')[0])
ba\bab
```

Escape sequence character sets

Commonly used character sets have predefined escape sequences:

- `\w` is similar to `[a-zA-Z0-9_]` for matching word characters (recall the definition for word boundaries)
- `\d` is similar to `[0-9]` for matching digit characters
- `\s` is similar to `[ \t\n\r\f\v]` for matching whitespace characters

These escape sequences can be used as standalone or inside a character class. Also, these would behave differently depending on flags used (covered in a later chapter). For now, as mentioned before, the examples and description will assume input made up of ASCII characters only.

```
>>> re.split(r'\d+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']
>>> re.findall(r'\d+', 'foo=5, bar=3; x=83, y=120')
['5', '3', '83', '120']

>>> ''.join(re.findall(r'\b\w', 'sea eat car rat eel tea'))
'secret'
>>> re.findall(r'[\w\s]+', 'tea sea-pit sit-lean bean')
['tea sea', 'pit sit', 'lean bean']
```

And negative logic strikes again, use `\W` , `\D` , and `\S` respectively for their negated character class.

```
>>> re.sub(r'\D+', r'-', 'Sample123string42with777numbers')
'-123-42-777-'

>>> re.sub(r'\W+', r'', 'foo=5, bar=3; x=83, y=120')
'foo5bar3x83y120'

>>> re.findall(r'\S+', '   1..3  \v\f  foo_baz 42\tzzz   \r\n1-2-3  ')
['1..3', 'foo_baz', '42', 'zzz', '1-2-3']
```

This chapter focussed on how to use and create custom placeholders for limited set of characters. Grouping and character classes can be considered as two levels of abstractions. On the one hand, you can have character sets inside `[]` and on the other, you can have multiple alternations grouped inside `()` including character classes. As anchoring and quantifiers can be applied to both these abstractions, you can begin to see how regular expressions is considered a mini-programming language. In coming chapters, you'll even see how to negate groupings similar to negated character class in certain scenarios.

Exercises

a) Delete all pair of parentheses, unless they contain a parentheses character.

```
>>> str1 = 'def factorial()'
>>> str2 = 'a/b(division) + c%d(#modulo) - (e+(j/k-3)*4)'
>>> str3 = 'Hi there(greeting). Nice day(a(b)'

>>> remove_parentheses = re.compile()      ##### add your solution here
>>> remove_parentheses.sub('', str1)
'def factorial'
>>> remove_parentheses.sub('', str2)
'a/b + c%d - (e+*4)'
```

```
>>> remove_parentheses.sub('', str3)
'Hi there. Nice day(a'
```

b) Extract all hex character sequences, with optional prefix. Match the characters case insensitively, and the sequences shouldn't be surrounded by other word characters.

```
>>> hex_seq = re.compile()        ##### add your solution here

>>> str1 = '128A foo 0xfe32 34 0xbar'
##### add your solution here
['128A', '0xfe32', '34']

>>> str2 = '0XDEADBEEF lace 0x0ff1ce bad'
##### add your solution here
['0XDEADBEEF', '0x0ff1ce', 'bad']
```

c) Output True/False depending upon input string containing any number sequence that is greater than `624`

```
>>> str1 = 'hi0000432abcd'
##### add your solution here
False

>>> str2 = '42_624 0512'
##### add your solution here
False

>>> str3 = '3.14 96 2 foo1234baz'
##### add your solution here
True
```

d) Split the given strings based on consecutive sequence of digit or whitespace characters.

```
>>> str1 = 'lion \t Ink32onion Nice'
>>> str2 = '**1\f2\n3star\t7 77\r**'
>>> expr = re.compile()       ##### add your solution here

>>> expr.split(str1)
['lion', 'Ink', 'onion', 'Nice']
>>> expr.split(str2)
['**', 'star', '**']
```

Groupings and backreferences

You've been patiently hearing more awesome stuff to come regarding groupings. Well, here they come in various forms. And some more will come in later chapters!

First up, saving (i.e. capturing) RE to use it later, similar to variables and functions in a programming language. You have already seen how to use Match object to refer to text captured by groups. In a similar manner, you can use backreference `\N` where `N` is the capture group you want. Backreferences can be used within the RE definition itself as well as in replacement section whereas saved Match objects can be used in later instructions.

In replacement section, use:

- `\1` , `\2` up to `\99` to refer to the corresponding capture group
  - provided there are no digit characters after
  - sequence of `\` and 3 digits will be interpreted as octal value
- `\g<1>` , `\g<2>` and so on (not limited to 99) to refer to the corresponding capture group
  - this also helps to avoid ambiguity, for ex: if you need to reference `\1` followed by other digit characters
- `\g<0>` to refer to entire matched portion, similar to index `0` of Match objects
  - `\0` cannot be used because numbers starting with `0` are treated as octal value

```
# remove square brackets that surround digit characters
>>> re.sub(r'\[(\d+)\]', r'\1', '[52] apples and [31] mangoes')
'52 apples and 31 mangoes'
# replace __ with _ and delete _ if it is alone
>>> re.sub(r'(_)?_', r'\1', '_foo_ __123__ _baz_')
'foo _123_ baz'

# add something around the matched strings
>>> re.sub(r'\d+', r'(\g<0>0)', '52 apples and 31 mangoes')
'(520) apples and (310) mangoes'
# note the use of count flag
# otherwise empty string matching with * will come into play
>>> re.sub(r'.*', r'Hi. \g<0>. Have a nice day', 'Hello world', count=1)
'Hi. Hello world. Have a nice day'

# swap words that are separated by a comma
>>> re.sub(r'(\w+),(\w+)', r'\2,\1', 'a,b 42,24')
'b,a 24,42'
```

Here's some examples for using backreferences within RE definition.

```
# whole words that have at least one consecutive repeated character
>>> words = ['effort', 'flee', 'facade', 'oddball', 'rat', 'tool']
>>> [w for w in words if re.search(r'\b\w*(\w)\1\w*\b', w)]
['effort', 'flee', 'oddball', 'tool']

# remove any number of consecutive duplicate words separated by space
# quantifiers can be applied to backreferences too!
>>> re.sub(r'\b(\w+)( \1)+\b', r'\1', 'a a a walking for for a cause')
'a walking for a cause'
```

Non-capturing groups

Grouping has many uses like applying quantifier on a RE portion, creating terse RE by factoring common portions and so on. It also affects behavior of functions like `re.findall` and `re.split` .

```
# without capture group
>>> re.split(r'\d+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']

# to include the matching delimiter strings as well in the output
>>> re.split(r'(\d+)', 'Sample123string42with777numbers')
```

```
['Sample', '123', 'string', '42', 'with', '777', 'numbers']

# optional argument maxsplit can be used to specify no. of splits
# setting to 1 gives behavior like partition string method
>>> re.split(r'(1*2)', '3111111111125111142', maxsplit=1)
['3', '11111111112', '5111142']
```

When backreferencing is not required, you can use a non-capturing group to avoid behavior change of `re.findall` and `re.split`. It also helps to avoid keeping a track of capture group numbers when that particular group is not needed for backreferencing. The syntax is `(?:RE)` to define a non-capturing group. More such special groups starting with `(?` syntax will be discussed later on.

```
# normal capture group will hinder ability to get whole match
# non-capturing group to the rescue
>>> re.findall(r'\b\w*(?:st|in)\b', 'cost akin more east run against')
['cost', 'akin', 'east', 'against']

# capturing wasn't needed here, only common grouping and quantifier
>>> re.split(r'hand(?:y|ful)?', '123hand42handy777handful500')
['123', '42', '777', '500']

# with normal grouping, need to keep track of all the groups
>>> re.sub(r'\A(([^,]+,){3})([^,]+)', r'\1(\3)', '1,2,3,4,5,6,7', count=1)
'1,2,3,(4),5,6,7'
# using non-capturing groups, only relevant groups have to be tracked
>>> re.sub(r'\A((?:[^,]+,){3})([^,]+)', r'\1(\2)', '1,2,3,4,5,6,7', count=1)
'1,2,3,(4),5,6,7'
```

However, there are situations where capture groups cannot be avoided. In such cases, you'd need to manually work with Match objects to get desired results.

```
>>> words = 'effort flee facade oddball rat tool'
# whole words containing at least one consecutive repeated character
>>> repeat_char = re.compile(r'\b\w*(\w)\1\w*\b')

# () in findall will only return text matched by capture groups
>>> repeat_char.findall(words)
['f', 'e', 'l', 'o']

# finditer will allow to use Match object for each match
>>> m_iter = repeat_char.finditer(words)
>>> [m[0] for m in m_iter]
['effort', 'flee', 'oddball', 'tool']
```

Named capture groups

RE can get cryptic and difficult to maintain, even for seasoned programmers. There are a few constructs to help add clarity. One such is naming the capture groups and using that name for backreferencing instead of plain numbers. The syntax is `(?P<name>RE)` for naming the capture groups. The name used should be a valid Python identifier. Use `'name'` for Match objects, `\g<name>` in replacement section and `(?P=name)` in RE definition for backreferencing. These will still behave as normal capture groups, so `\N` or `\g<N>` numbering can be used as well.

```
# giving names to first and second captured words
>>> re.sub(r'(?P<fw>\w+),(?P<sw>\w+)', r'\g<sw>,\g<fw>', 'a,b 42,24')
'b,a 24,42'

>>> sentence = 'I bought an apple'
>>> m = re.search(r'(?P<fruit>\w+)\Z', sentence)
>>> m[1]
'apple'
>>> m['fruit']
```

```
'apple'
>>> m.group('fruit')
'apple'
```

Subexpression calls

It may be obvious, but it should be noted that backreference will provide the string that was matched, not the RE that was inside the capture group. For example, if `([0-9][a-f])` matches `3b`, then backreferencing will give `3b` and not any other valid match of RE like `8f`, `0a` etc. This is akin to how variables behave in programming, only the result of expression stays after variable assignment, not the expression itself.

The `regex` module provides a way to refer to the expression itself, using `(?1)`, `(?2)` etc. This is applicable only in RE definition, not in replacement sections. This behavior is similar to function call, and like functions this can simulate recursion as well (will be discussed later).

```
>>> import re, regex
>>> row = 'today,2008-03-24,food,2012-08-12,nice,5632'

# with re module and manually repeating the pattern
>>> re.search(r'\d{4}-\d{2}-\d{2}.*\d{4}-\d{2}-\d{2}', row)[0]
'2008-03-24,food,2012-08-12'

# with regex module and subexpression calling
>>> regex.search(r'(\d{4}-\d{2}-\d{2}).*(?1)', row)[0]
'2008-03-24,food,2012-08-12'
```

Named capture group can be used as well and called using `(?&name)` syntax:

```
>>> import regex
>>> row = 'today,2008-03-24,food,2012-08-12,nice,5632'

>>> regex.search(r'(?P<date>\d{4}-\d{2}-\d{2}).*(?&date)', row)[0]
'2008-03-24,food,2012-08-12'
```

This chapter covered many more features related to grouping - backreferencing to get both variable and function like behavior, and naming the groups to add clarity. When backreference is not needed for a particular group, use non-capturing group.

Exercises

a) The given string has fields separated by `:` and each field has a floating point number followed by a `,` and a string. If the floating point number has only one digit precision, append `0` and swap the strings separated by `,` for that particular field.

```
>>> row = '3.14,hi:42.5,bye:1056.1,cool:00.9,fool'
##### add your solution here
'3.14,hi:bye,42.50:cool,1056.10:fool,00.90'
```

b) Count the number of words that have at least two consecutive repeated alphabets, for ex: words like `stillness` and `Committee` but not words like `root` or `readable` or `rotational`. Consider word to be as defined in regular expression parlance and word split across two lines as two different words.

```
>>> import urllib.request
>>> scarlet_pimpernel_link = r'https://www.gutenberg.org/cache/epub/60/pg60.txt'
>>> word_expr = re.compile()      ##### add your solution here
>>> count = 0
>>> with urllib.request.urlopen(scarlet_pimpernel_link) as ip_file:
...     for line in ip_file:
...         for word in re.findall(rb'\w+', line):
...             if word_expr.search(word):
...                 count += 1
...
```

```
>>> print(count)
219
```

c) Convert the given **markdown** headers to corresponding **anchor** tag. Consider the input to start with one or more `#` characters followed by space and word characters. The `name` attribute is constructed by converting the header to lowercase and replacing spaces with hyphens. Can you do it without using a capture group?

```
>>> header1 = '# Regular Expressions'
>>> header2 = '## Compiling regular expressions'

##### add your solution here for header1
'# <a name="regular-expressions"></a>Regular Expressions'
##### add your solution here for header2
'## <a name="compiling-regular-expressions"></a>Compiling regular expressions'
```

d) Convert the given **markdown** anchors to corresponding **hyperlinks**.

```
>>> anchor1 = '# <a name="regular-expressions"></a>Regular Expressions'
>>> anchor2 = '## <a name="subexpression-calls"></a>Subexpression calls'

##### add your solution here for anchor1
'[Regular Expressions](#regular-expressions)'
##### add your solution here for anchor2
'[Subexpression calls](#subexpression-calls)'
```

e) Use appropriate regular expression function to get the expected output for given string.

```
>>> str1 = 'price_42 roast:\t\n:-ice==cat\neast'
##### add your solution here
['price_42', ' ', 'roast', ':\t\n:-', 'ice', '==', 'cat', '\n', 'east']
```

Lookarounds

Having seen how to create custom character classes and various avatars of groupings, it is time for learning how to create custom anchors and add conditions to a pattern within RE definition. These assertions are also known as **zero-width patterns** because they add restrictions similar to anchors and are not part of matched portions. Also, you will learn how to negate a grouping similar to negated character sets.

Negative lookarounds

Lookaround assertions can be added to a pattern in two ways - as a prefix known as **lookbehind** and as a suffix known as **lookahead**. Syntax wise, these two ways are differentiated by adding a `<` for the lookbehind version. Negative lookaround uses `!` to indicate negated logic. The complete syntax looks like:

- `(?!RE)` for negative lookahead assertion
- `(?<!RE)` for negative lookbehind assertion

As mentioned earlier, lookarounds are not part of matched portions and do not capture the matched text.

```
# change 'foo' only if it is not followed by a digit character
# note that end of string satisfies the given assertion
# 'foofoo' has two matches as the assertion doesn't consume characters
>>> re.sub(r'foo(?!\d)', r'baz', 'hey food! foo42 foot5 foofoo')
'hey bazd! foo42 bazt5 bazbaz'

# change 'foo' only if it is not preceded by _
# note how 'foo' at start of string is matched as well
>>> re.sub(r'(?<!_)foo', r'baz', 'foo _foo 42foofoo')
'baz _foo 42bazbaz'

# overlap example
# the final _ was replaced as well as played a part in assertion
>>> re.sub(r'(?<!_)foo.', r'baz', 'food _fool 42foo_foot')
'baz _fool 42bazfoot'
```

Can be mixed with already existing anchors and other features to define truly powerful restrictions:

```
# change whole word only if it is not preceded by : or -
>>> re.sub(r'(?<![:-])\b\w+\b', r'X', ':cart <apple -rest ;tea')
':cart <X -rest ;X'

# add space to word boundaries, but not at start or end of string
# similar to: re.sub(r'\b', r' ', 'foo_baz=num1+35*42/num2').strip()
>>> re.sub(r'(?<!\A)\b(?!\Z)', r' ', 'foo_baz=num1+35*42/num2')
'foo_baz = num1 + 35 * 42 / num2'
```

Positive lookarounds

Positive lookaround syntax uses `=` similar to `!` for negative lookaround. The complete syntax looks like:

- `(?=RE)` for positive lookahead assertion
- `(?<=RE)` for positive lookbehind assertion

```
# extract digits only if it is followed by ,
# note that end of string doesn't qualify as this is positive assertion
>>> re.findall(r'\d+(?=,)', '42 foo-5, baz3; x-83, y-20: f12')
['5', '83']
# extract digits only if it is preceded by - and followed by ; or :
>>> re.findall(r'(?<=-)\d+(?=[:;])', '42 foo-5, baz3; x-83, y-20: f12')
['20']
```

Lookarounds are quite handy in dealing with field based processing:

```
# except first and last fields
>>> re.findall(r'(?<=,)[^,]+(?=,)', '1,two,3,four,5')
['two', '3', 'four']

# replace empty fields with nil
# note that in this case, order of lookbehind and lookahead doesn't matter
>>> re.sub(r'(?<![^,])(?![^,])', r'NA', ',1,,,two,3,,,')
'NA,1,NA,NA,two,3,NA,NA,NA'
```

Even though lookarounds are not part of matched portions, capture groups can be used inside them.

```
>>> print(re.sub(r'(\S+\s+)(?=(\S+)\s)', r'\1\2\n', 'a b c d e'))
a b
b c
c d
d e

# and of course, use non-capturing group where needed
>>> re.findall(r'(?<=(po|ca)re)\d+', 'pore42 car3 pare7 care5')
['po', 'ca']
>>> re.findall(r'(?<=(?:po|ca)re)\d+', 'pore42 car3 pare7 care5')
['42', '5']
```

AND conditional

As promised earlier, lookarounds can be used to construct AND conditional.

```
>>> words = ['sequoia', 'subtle', 'questionable', 'exhibit', 'equation']

# words containing 'b' and 'e' and 't' in any order
# same as: r'b.*e.*t|b.*t.*e|e.*b.*t|e.*t.*b|t.*b.*e|t.*e.*b'
>>> [w for w in words if re.search(r'(?=.*b)(?=.*e).*t', w)]
['subtle', 'questionable', 'exhibit']
# words containing all lowercase vowels in any order
>>> [w for w in words if re.search(r'(?=.*a)(?=.*e)(?=.*i)(?=.*o).*u', w)]
['sequoia', 'questionable', 'equation']
```

Variable length lookbehind

When using lookbehind assertion (either positive or negative), the RE inside the assertion cannot imply matching variable length of text. Using fixed length quantifier is allowed. Alternations of different lengths, even if the different alternations are of fixed length, are not allowed. Here's some examples to clarify these points:

```
# allowed
>>> re.findall(r'(?<=(?:po|ca)re)\d+', 'pore42 car3 pare7 care5')
['42', '5']
>>> re.findall(r'(?<=\b[a-z]{4})\d+', 'pore42 car3 pare7 care5')
['42', '7', '5']

# not allowed
>>> re.findall(r'(?<!car|pare)\d+', 'pore42 car3 pare7 care5')
re.error: look-behind requires fixed-width pattern
>>> re.findall(r'(?<=\b[a-z]+)\d+', 'pore42 car3 pare7 care5')
re.error: look-behind requires fixed-width pattern
>>> re.sub(r'(?<=\A|,)(?=,|\Z)', r'NA', ',1,,,two,3,,,')
re.error: look-behind requires fixed-width pattern
```

Variable length lookbehind can be addressed in multiple ways using the `regex` module. Some of the variable length positive lookbehind cases can be simulated by using `\K` as a suffix to the RE that is needed as lookbehind assertion.

```
>>> import regex

# similar to: r'(?<=\b\w)\w*\W*'
# text matched before \K won't be replaced
>>> regex.sub(r'\b\w\K\w*\W*', r'', 'sea eat car rat eel tea')
'secret'

# replace only 3rd occurrence of 'cat'
>>> regex.sub(r'(cat.*?){2}\Kcat', r'X', 'cat scatter cater scat', count=1)
'cat scatter Xer scat'
```

If `\K` doesn't work out for some reason, the `regex` module allows using variable length lookbehind as is.

```
>>> regex.findall(r'(?<=\b[a-z]+)\d+', 'pore42 car3 pare7 care5')
['42', '3', '7', '5']

>>> regex.sub(r'(?<=\A|,)(?=,|\Z)', r'NA', ',1,,,two,3,,,')
'NA,1,NA,NA,two,3,NA,NA,NA'

>>> regex.sub(r'(?<=(cat.*?){2})cat', r'X', 'cat scatter cater scat', count=1)
'cat scatter Xer scat'
```

Here's some variable length negative lookbehind examples:

```
>>> regex.findall(r'(?<!car|pare)\d+', 'pore42 car3 pare7 care5')
['42', '5']

# match 'dog' only if it is not preceded by 'cat'
>>> bool(regex.search(r'(?<!cat.*)dog', 'fox,cat,dog,parrot'))
False

# match 'dog' only if it is not preceded by 'parrot'
>>> bool(regex.search(r'(?<!parrot.*)dog', 'fox,cat,dog,parrot'))
True
```

Negated groups

Variable length negative lookbehind can also be simulated using negative lookahead (which doesn't have restriction on variable length) inside a grouping and applying quantifier to match characters one by one. This also showcases how grouping can be negated in certain cases.

```
# note the use of \A anchor to force matching all characters up to 'dog'
# also note that regex module is not needed here
>>> bool(re.search(r'\A((?!cat).)*dog', 'fox,cat,dog,parrot'))
False
>>> bool(re.search(r'\A((?!parrot).)*dog', 'fox,cat,dog,parrot'))
True

# easier to understand by checking matched portion
>>> re.search(r'\A((?!cat).)*', 'fox,cat,dog,parrot')[0]
'fox,'
>>> re.search(r'\A((?!parrot).)*', 'fox,cat,dog,parrot')[0]
'fox,cat,dog,'
>>> re.search(r'\A((?!(.)\2).)*', 'fox,cat,dog,parrot')[0]
'fox,cat,dog,pa'
```

As lookarounds do not consume characters, you cannot use variable length lookbehind (assuming `regex` module) between two patterns. Use negated groups instead.

```
# match if 'do' is not there between 'at' and 'par'
>>> bool(re.search(r'at((?!do).)*par', 'fox,cat,dog,parrot'))
False
```

```
# match if 'go' is not there between 'at' and 'par'
>>> bool(re.search(r'at((?!go).)*par', 'fox,cat,dog,parrot'))
True
>>> re.search(r'at((?!go).)*par', 'fox,cat,dog,parrot')[0]
'at,dog,par'
```

In this chapter, you learnt how to use lookarounds to create custom restrictions and also how to use negated grouping. With this, most of the powerful features of regular expressions have been covered. The special groupings seem never ending though, there's some more of them in coming chapters!!

Exercises

a) Remove leading and trailing whitespaces from all the individual fields of these csv strings.

```
>>> csv1 = ' comma  ,separated ,values '
>>> csv2 = 'good bad,nice  ice  , 42 , ,   stall   small'

>>> remove_whitespace = re.compile()     ##### add your solution here
>>> remove_whitespace.sub('', csv1)
'comma,separated,values'
>>> remove_whitespace.sub('', csv2)
'good bad,nice  ice,42,,stall   small'
```

b) Filter all elements that satisfy all of these rules:

- should have at least two alphabets
- should have at least 3 digits
- should have at least one special character among `%` or `*` or `#` or `$`
- should not end with a whitespace character

```
>>> pwds = ['hunter2', 'F2H3u%9', '*X3Yz3.14\t', 'r2_d2_42', 'A $B C1234']
##### add your solution here
['F2H3u%9', 'A $B C1234']
```

c) Match strings if it doesn't contain whitespace or the string `error` between the strings `qty` and `price`

```
>>> str1 = '23,qty,price,42'
>>> str2 = 'qty price,oh'
>>> str3 = '3.14,qty,6,errors,9,price,3'
>>> str4 = 'qty-6,apple-56,price-234'

>>> neg = re.compile()       ##### add your solution here
>>> bool(neg.search(str1))
True
>>> bool(neg.search(str2))
False
>>> bool(neg.search(str3))
False
>>> bool(neg.search(str4))
True
```

Flags

Just like options change the default behavior of commands used from a terminal, flags are used to change aspects of RE. The **Anchors** chapter already introduced one of them. Flags can be applied to entire RE using `flags` optional argument or to a particular portion of RE using special groups. And both of these forms can be mixed up as well. In regular expression parlance, flags are also known as **modifiers**.

First up, the flag to ignore case while matching alphabets. When `flags` argument is used, this can be specified as `re.I` or `re.IGNORECASE` constants.

```
>>> bool(re.search(r'cat', 'Cat'))
False
>>> bool(re.search(r'cat', 'Cat', flags=re.IGNORECASE))
True

>>> re.findall(r'c.t', 'Cat cot CATER ScUtTLe', flags=re.I)
['Cat', 'cot', 'CAT', 'cUt']

# without flag, you need to use: r'[a-zA-Z]+'
# with flag, can also use: r'[A-Z]+'
>>> re.findall(r'[a-z]+', 'Sample123string42with777numbers', flags=re.I)
['Sample', 'string', 'with', 'numbers']
```

Use `re.S` or `re.DOTALL` to allow `.` metacharacter to match newline character as well.

```
# by default, the . metacharacter doesn't match newline
>>> re.sub(r'the.*ice', r'X', 'Hi there\nHave a Nice Day')
'Hi there\nHave a Nice Day'

# re.S flag will allow newline character to be matched as well
>>> re.sub(r'the.*ice', r'X', 'Hi there\nHave a Nice Day', flags=re.S)
'Hi X Day'

# multiple flags can be combined using bitwise OR operator
>>> re.sub(r'the.*day', r'Bye', 'Hi there\nHave a Nice Day', flags=re.S|re.I)
'Hi Bye'
```

As seen earlier, `re.M` or `re.MULTILINE` flag would allow `^` and `$` anchors to match line wise instead of whole string.

```
# check if any line in the string starts with 'top'
>>> bool(re.search(r'^top', "hi hello\ntop spot", flags=re.M))
True

# check if any line in the string ends with 'ar'
>>> bool(re.search(r'ar$', "spare\npar\ndare", flags=re.M))
True
```

The `re.X` or `re.VERBOSE` flag is another provision like the named capture groups to help add clarity to RE definitions. This flag allows to use literal whitespaces for aligning purposes and add comments after the `#` character to break down complex RE into multiple lines with comments.

```
# same as: rex = re.compile(r'\A((?:[^,]+,){3})([^,]+)')
# note the use of triple quoted string
>>> rex = re.compile(r'''
...         \A(                 # group-1, captures first 3 columns
...             (?:[^,]+,){3}   # non-capturing group to get the 3 columns
...           )
...         ([^,]+)             # group-2, captures 4th column
...         ''', flags=re.X)

>>> rex.sub(r'\1(\2)', '1,2,3,4,5,6,7', count=1)
'1,2,3,(4),5,6,7'
```

For precise definition, here's the relevant quote from documentation:

> Whitespace within the pattern is ignored, except when in a character class, or when preceded by an unescaped backslash, or within tokens like `*?` , `(?:` or `(?P<...>` . When a line contains a `#` that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such `#` through the end of the line are ignored.

```
>>> bool(re.search(r't a', 'cat and dog', flags=re.X))
False
>>> bool(re.search(r't\ a', 'cat and dog', flags=re.X))
True
>>> bool(re.search(r't[ ]a', 'cat and dog', flags=re.X))
True
>>> bool(re.search(r't\x20a', 'cat and dog', flags=re.X))
True

>>> re.search(r'a#b', 'foo a#b 123', flags=re.X)[0]
'a'
>>> re.search(r'a\#b', 'foo a#b 123', flags=re.X)[0]
'a#b'
```

Comments can also be added using `(?#comment)` special group.

```
>>> rex = re.compile(r'\A((?:[^,]+,){3})(?#3-cols)([^,]+)(?#4th-col)')

>>> rex.sub(r'\1(\2)', '1,2,3,4,5,6,7', count=1)
'1,2,3,(4),5,6,7'
```

To apply flags to specific portions of RE, specify them inside a special grouping syntax. This will override the flags applied to entire RE definitions, if any. The syntax variations are:

- `(?flags:RE)` will apply flags only for this RE portion
- `(?-flags:RE)` will negate flags only for this RE portion
- `(?flags-flags:RE)` will apply and negate particular flags only for this RE portion
- `(?flags)` will apply flags for whole RE definition, can only be specified at start of RE definition
  - if anchors are needed, they should be specified after these flags

In these ways, flags can be specified precisely only where it is needed. The flags are to be given as single letter lowercase version of short form constants - for ex: `i` for `re.I` and so on, except `L` for `re.L` or `re.LOCALE` (will be discussed later). And as can be observed from below examples, these are not capture groups.

```
# case-sensitive for whole RE definition
>>> re.findall(r'Cat[a-z]*\b', 'Cat SCatTeR CATER cAts')
['Cat']
# case-insensitive only for '[a-z]*' portion
>>> re.findall(r'Cat(?i:[a-z]*)\b', 'Cat SCatTeR CATER cAts')
['Cat', 'CatTeR']

# case-insensitive for whole RE definition using flags argument
>>> re.findall(r'Cat[a-z]*\b', 'Cat SCatTeR CATER cAts', flags=re.I)
['Cat', 'CatTeR', 'CATER', 'cAts']
# case-insensitive for whole RE definition using special group
>>> re.findall(r'(?i)Cat[a-z]*\b', 'Cat SCatTeR CATER cAts')
['Cat', 'CatTeR', 'CATER', 'cAts']
# case-sensitive only for 'Cat' portion
>>> re.findall(r'(?-i:Cat)[a-z]*\b', 'Cat SCatTeR CATER cAts', flags=re.I)
['Cat', 'CatTeR']
```

This chapter showed some of the flags that can be used to change default behavior of RE definition. And more special groupings were covered.

Exercises

a) Delete from the string `start` if it is at beginning of a line up to the next occurrence of the string `end` at end of a line. Match these keywords irrespective of case.

```
>>> para = '''\
... good start
... start working on that
... project you always wanted
... to, do not let it end
... hi there
... start and end the end
... 42
... Start and try to
... finish the End
... bye'''

>>> expr = re.compile()       ##### add your solution here
>>> print(expr.sub('', para))
good start

hi there

42

bye
```

b) Explore what the `re.DEBUG` flag does. Here's some examples, check their output:

- `re.compile(r'\Aden|ly\Z', flags=re.DEBUG)`
- `re.compile(r'\b(0x)?[\da-f]+\b', flags=re.DEBUG)`
- `re.compile(r'\b(?:0x)?[\da-f]+\b', flags=re.I|re.DEBUG)`

Unicode

So far in the book, all examples were meant for strings made up of ASCII characters only. However, `re` module matching is Unicode by default. See docs.python: Unicode for a tutorial on Unicode support in Python.

Flags can be used to override the default setting. For example, the `re.A` or `re.ASCII` flag will change `\b`, `\w`, `\d`, `\s` and their opposites to match only ASCII characters. Use `re.L` or `re.LOCALE` to work based on locale settings for bytes data type.

```
# \w is Unicode aware
>>> re.findall(r'\w+', 'fox:αλεπού')
['fox', 'αλεπού']

# restrict matching to only ASCII characters
>>> re.findall(r'\w+', 'fox:αλεπού', flags=re.A)
['fox']
# or, explicitly define the characters to match using character class
>>> re.findall(r'[a-zA-Z0-9_]+', 'fox:αλεπού')
['fox']
```

However, the four characters shown below are also matched when `re.I` is used without `re.A`

```
>>> bool(re.search(r'[a-zA-Z]', 'İıſK'))
False

>>> re.search(r'[a-z]+', 'İıſK', flags=re.I)[0]
'İıſK'

>>> bool(re.search(r'[a-z]', 'İıſK', flags=re.I|re.A))
False
```

Unicode character sets

Similar to named character classes and escape sequences, the `regex` module supports `\p{}` construct that offers various predefined sets to work with Unicode strings. See regular-expressions: Unicode for details.

```
# extract all consecutive letters
>>> regex.findall(r'\p{L}+', 'fox:αλεπού,eagle:αετός')
['fox', 'αλεπού', 'eagle', 'αετός']
# extract all consecutive Greek letters
>>> regex.findall(r'\p{Greek}+', 'fox:αλεπού,eagle:αετός')
['αλεπού', 'αετός']

# extract all words
>>> regex.findall(r'\p{Word}+', 'φοο12,βτ_4,foo')
['φοο12', 'βτ_4', 'foo']

# delete all characters other than letters
# \p{^L} can also be used instead of \P{L}
>>> regex.sub(r'\P{L}+', r'', 'φοο12,βτ_4,foo')
'φοοβτfoo'
```

For generic Unicode character ranges, specify 4-hexdigits codepoint using `\u` or 8-hexdigits codepoint using `\U`

```
# to get codepoints for ASCII characters
>>> [hex(ord(c)) for c in 'fox']
['0x66', '0x6f', '0x78']
# to get codepoints for Unicode characters
>>> [c.encode('unicode_escape') for c in 'αλεπού']
[b'\\u03b1', b'\\u03bb', b'\\u03b5', b'\\u03c0', b'\\u03bf', b'\\u03cd']
>>> [c.encode('unicode_escape') for c in 'İıſK']
[b'\\u0130', b'\\u0131', b'\\u017f', b'\\u212a']
```

```
# character range example using \u
# all english lowercase letters
>>> re.findall(r'[\u0061-\u007a]+', 'fox:αλεπού,eagle:αετός')
['fox', 'eagle']
```

A comprehensive discussion on RE usage with Unicode characters is out of scope for this book. Resources like regular-expressions: unicode and Programmers introduction to Unicode are recommended for further study.

Exercises

a) Output `True` or `False` depending on input string made up of ASCII characters or not. Consider the input to be non-empty strings and any character that isn't part of 7-bit ASCII set should give `False`

```
>>> str1 = '123–456'
>>> str2 = 'good food'
>>> str3 = 'happy learning!'
>>> str4 = 'İıfK'

##### add your solution here for str1
False
##### add your solution here for str2
False
##### add your solution here for str3
True
##### add your solution here for str4
False
```

Miscellaneous

This chapter will cover some more features and useful tricks. Except first two sections, rest are all features provided by `regex` module.

Using dict

Using a function in replacement section, you can specify a `dict` variable to determine the replacement string based on the matched text.

```
# one to one mappings
>>> d = { '1': 'one', '2': 'two', '4': 'four' }
>>> re.sub(r'[124]', lambda m: d[m[0]], '9234012')
'9two3four0onetwo'

# if the matched text doesn't exist as a key, default value will be used
>>> re.sub(r'\d', lambda m: d.get(m[0], 'X'), '9234012')
'XtwoXfourXonetwo'
```

For swapping two or more portions without using intermediate result, using a `dict` is recommended.

```
>>> swap = { 'cat': 'tiger', 'tiger': 'cat' }
>>> words = 'cat tiger dog tiger cat'

# replace word if it exists as key, else leave it as is
>>> re.sub(r'\w+', lambda m: swap.get(m[0], m[0]), words)
'tiger cat dog cat tiger'

# or, build the alternation list manually for simple cases
>>> re.sub(r'cat|tiger', lambda m: swap[m[0]], words)
'tiger cat dog cat tiger'
```

For `dict` that have many entries and likely to undergo changes during development, building alternation list manually is not a good choice. Also, recall that as per precedence rules, longest length string should come first.

```
>>> d = { 'hand': 1, 'handy': 2, 'handful': 3, 'a^b': 4 }

# take care of metacharacter escaping first
>>> words = [re.escape(k) for k in d.keys()]
# build alternation list
>>> '|'.join(sorted(words, key=len, reverse=True))
'handful|handy|hand|a\\^b'
# add anchors and flags as needed to construct final RE
```

re.subn

The `re.subn` function returns a tuple - modified string after substitution and number of substitutions made. This can be used to perform conditional operations based on whether the substitution was successful. Or, the value of count itself may be needed for solving the given problem.

```
>>> word = 'coffining'
# recursively delete 'fin'
>>> while True:
...     word, cnt = re.subn(r'fin', r'', word)
...     if cnt == 0:
...         break
...
>>> word
'cog'
```

Here's an example that won't work if greedy quantifier is used instead of possessive quantifier.

```
>>> row = '421,foo,2425,42,5,foo,6,6,42'

# lookarounds used to ensure start/end of column matching
# possessive quantifier used to ensure partial column is not captured
# if a column has same text as another column, the latter column is deleted
>>> while True:
...     row, cnt = regex.subn(r'(?<=\A|,)([^,]++).*\K,\1(?=,|\Z)', r'', row)
...     if cnt == 0:
...         break
...
>>> row
'421,foo,2425,42,5,6'
```

\G anchor

The `\G` anchor (provided by `regex` module) restricts matching from start of string like the `\A` anchor. In addition, after a match is done, ending of that match is considered as the new anchor location. This process is repeated again and continues until the given RE fails to match (assuming multiple matches with `sub`, `findall` etc).

```
>>> import regex

# all non-whitespace characters from start of string
>>> regex.findall(r'\G\S', '123-87-593 42 foo')
['1', '2', '3', '-', '8', '7', '-', '5', '9', '3']
>>> regex.sub(r'\G\S', r'*', '123-87-593 42 foo')
'********** 42 foo'

# all digits and optional hyphen combo from start of string
>>> regex.findall(r'\G\d+-?', '123-87-593 42 foo')
['123-', '87-', '593']
>>> regex.sub(r'\G(\d+)(-?)', r'(\1)\2', '123-87-593 42 foo')
'(123)-(87)-(593) 42 foo'

# all word characters from start of string
# only if it is followed by word character
>>> regex.sub(r'\G\w(?=\w)', r'\g<0>:', 'cat12 bat pin')
'c:a:t:1:2 bat pin'

# all lowercase alphabets or space from start of string
>>> regex.sub(r'\G[a-z ]', r'(\g<0>)', 'par tar-den hen-food mood')
'(p)(a)(r)( )(t)(a)(r)-den hen-food mood'
```

Recursive matching

The subexpression call special group was introduced as analogous to function call. And in typical function fashion, it does support recursion. Useful to match nested patterns, which is usually not recommended to be done with regular expressions. Indeed, if you are looking to parse file formats like html, xml, json, csv, etc - use a proper parser library. But for some cases, a parser might not be available and using RE might be simpler than writing a parser from scratch.

First up, a RE to match a set of parentheses that is not nested (termed as level-one RE for reference).

```
# note the use of possessive quantifier
>>> eqn0 = 'a + (b * c) - (d / e)'
>>> regex.findall(r'\([^()]++\)', eqn0)
['(b * c)', '(d / e)']

>>> eqn1 = '((f+x)^y-42)*((3-g)^z+2)'
```

```
>>> regex.findall(r'\([^()]++\)', eqn1)
['(f+x)', '(3-g)']
```

Next, matching a set of parentheses which may optionally contain any number of non-nested sets of parentheses (termed as level-two RE for reference).

```
>>> eqn1 = '((f+x)^y-42)*((3-g)^z+2)'
# note the use of non-capturing group
>>> regex.findall(r'\((?:[^()]++|\([^()]++\))++\)', eqn1)
['((f+x)^y-42)', '((3-g)^z+2)']

>>> eqn2 = 'a + (b) + ((c)) + (((d)))'
>>> regex.findall(r'\((?:[^()]++|\([^()]++\))++\)', eqn2)
['(b)', '((c))', '((d))']
```

That looks very cryptic. Better to use `re.X` flag for clarity as well as for comparing against the recursive version. Breaking down the RE, you can see `(` and `)` have to be matched literally. Inside that, valid string is made up of either non-parentheses characters or a non-nested parentheses sequence (level-one RE).

```
>>> lvl2 = regex.compile('''
...          \(              #literal (
...            (?:           #start of non-capturing group
...             [^()]++      #non-parentheses characters
...             |            #OR
...             \([^()]++\)  #level-one RE
...            )++           #end of non-capturing group, 1 or more times
...          \)              #literal )
...          ''', flags=re.X)

>>> lvl2.findall(eqn1)
['((f+x)^y-42)', '((3-g)^z+2)']

>>> lvl2.findall(eqn2)
['(b)', '((c))', '((d))']
```

To recursively match any number of nested sets of parentheses, use a capture group and call it within the capture group itself. Since entire RE needs to be called here, you can use the default zeroth capture group (this also helps to avoid having to use `finditer`). Comparing with level-two RE, the only change is that `(?0)` is used instead of the level-one RE in the second alternation.

```
>>> lvln = regex.compile('''
...          \(           #literal (
...            (?:        #start of non-capturing group
...             [^()]++   #non-parentheses characters
...             |         #OR
...             (?0)      #recursive call
...            )++        #end of non-capturing group, 1 or more times
...          \)           #literal )
...          ''', flags=re.X)

>>> lvln.findall(eqn0)
['(b * c)', '(d / e)']

>>> lvln.findall(eqn1)
['((f+x)^y-42)', '((3-g)^z+2)']

>>> lvln.findall(eqn2)
['(b)', '((c))', '(((d)))']

>>> eqn3 = '(3+a) * ((r-2)*(t+2)/6) + 42 * (a(b(c(d(e)))))'
>>> lvln.findall(eqn3)
['(3+a)', '((r-2)*(t+2)/6)', '(a(b(c(d(e)))))']
```

Named character sets

A named character set is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]`, along with any other characters as needed. Using `[:^` instead of `[:` will negate the named character set. See regular-expressions: POSIX Bracket for full list, and refer to pypi: regex for notes on Unicode.

```
# similar to: r'\d+' or r'[0-9]+'
>>> regex.split(r'[[:digit:]]+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']
# similar to: r'[a-zA-Z]+'
>>> regex.findall(r'[[:alpha:]]+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']

# similar to: r'[\w\s]+'
>>> regex.findall(r'[[:word:][:space:]]+', 'tea sea-pit sit-lean bean')
['tea sea', 'pit sit', 'lean bean']
# similar to: r'\S+'
>>> regex.findall(r'[[:^space:]]+', 'tea sea-pit sit-lean bean')
['tea', 'sea-pit', 'sit-lean', 'bean']

# words not surrounded by punctuation characters
>>> regex.findall(r'(?<![[:punct:]])\b\w+\b(?![[:punct:]])', 'tie. ink eat;')
['ink']
```

Character class set operations

There are two versions provided by `regex` module - by default version `0` is used, which is meant for compatibility with `re` module. Many features, like set operations, require version `1` to be enabled. That can be done by assigning `regex.DEFAULT_VERSION` to `regex.VERSION1` (permanent) or using `(?V1)` flag (temporary). To get back the compatible version, use `regex.VERSION0` or `(?V0)`

Set operations can be applied inside character class between sets. Mostly used to get intersection or difference between two sets, where one/both of them is a character range or predefined character set. To aid in such definitions, you can use `[]` in nested fashion. The four operators, in increasing order of precedence, are:

- `||` union
- `~~` symmetric difference
- `&&` intersection
- `--` difference

Note: These set operations may get added to `re` module in future.

```
# [^aeiou] will match any non-vowel character
# which means space is also a valid character to be matched
>>> re.findall(r'\b[^aeiou]+\b', 'tryst glyph pity why')
['tryst glyph ', ' why']
# intersection or difference can be used here
# to get a positive definition of characters to match
>>> regex.findall(r'(?V1)\b[a-z&&[^aeiou]]+\b', 'tryst glyph pity why')
['tryst', 'glyph', 'why']

# [[a-l]~~[g-z]] is same as [a-fm-z]
>>> regex.findall(r'(?V1)\b[[a-l]~~[g-z]]+\b', 'gets eat top sigh')
['eat', 'top']

# remove all punctuation characters except . ! and ?
>>> para = '"Hi", there! How *are* you? All fine here.'
>>> regex.sub(r'(?V1)[[:punct:]--[.!?]]+', r'', para)
'Hi there! How are you? All fine here.'
```

Skipping matches

Sometimes, you want to change or extract all matches except particular matches. Usually, there are common characteristics between the two types of matches that makes it hard or impossible to define RE only for the required matches. For ex: changing field values unless it is a particular name, or perhaps don't touch double quoted values and so on. To use the skipping feature, define the matches to be ignored suffixed by `(*SKIP)(*FAIL)` and then define the matches required as part of alternation. `(*F)` can also be used instead of `(*FAIL)` .

```
# change lowercase words other than imp or rat
>>> words = 'tiger imp goat eagle rat'
>>> regex.sub(r'\b(?:imp|rat)\b(*SKIP)(*F)|[a-z]++', r'(\g<0>)', words)
'(tiger) imp (goat) (eagle) rat'

# change all commas other than those inside double quotes
>>> row = '1,"cat,12",nice,two,"dog,5"'
>>> regex.sub(r'"[^"]++"(*SKIP)(*F)|,', r'|', row)
'1|"cat,12"|nice|two|"dog,5"'
```

This is a miscellaneous chapter, not able to think of a good catchy summary to write. Here's a suggestion - write a summary in your own words based on notes you've made for this chapter.

Exercises

a) Count the maximum depth of nested braces for the given string. Unbalanced or wrongly ordered braces should return `-1`

```
>>> def max_nested_braces(ip):
##### add your solution here

>>> max_nested_braces('a*b')
0
>>> max_nested_braces('}a+b{')
-1
>>> max_nested_braces('a*b+{}')
1
>>> max_nested_braces('{{a+2}*{b+c}+e}')
2
>>> max_nested_braces('{{a+2}*{b+{c*d}}+e}')
3
>>> max_nested_braces('{{a+2}*{\n{b+{c*d}}+e*d}}')
4
>>> max_nested_braces('a*{b+c*{e*3.14}}}')
-1
```

b) Replace the string `par` with `spar` , `spare` with `extra` and `park` with `garden`

```
>>> str1 = 'apartment has a park'
##### add your solution here for str1
'aspartment has a garden'

>>> str2 = 'do you have a spare cable'
##### add your solution here for str2
'do you have a extra cable'

>>> str3 = 'write a parser'
##### add your solution here for str3
'write a sparser'
```

c) Read about `POSIX` flag from regex module documentation. Is the following code snippet showing the correct output?

```
>>> words = 'plink incoming tint winter in caution sentient'
>>> change = regex.compile(r'int|in|ion|ing|inco|inter|ink', flags=regex.POSIX)
```

```
>>> change.sub(r'X', words)
'plX XmX tX wX X cautX sentient'
```

d) For the given **markdown** file, replace all occurrences of the string `python` (irrespective of case) with the string `Python` . However, any match within code blocks that start with whole line ```` ```python ```` and end with whole line ```` ``` ```` shouldn't be replaced. Consider the input file to be small enough to fit memory requirements.

Refer to exercises folder for files required to solve this exercise.

```
>>> ip_str = open('sample.md', 'r').read()
>>> expr = regex.compile()      ##### add your solution here
>>> with open('sample_mod.md', 'w') as op_file:
...     op_file.write(expr.sub(lambda m: m[0].capitalize(), ip_str))
...
305
>>> assert open('sample_mod.md').read() == open('expected.md').read()
```

Gotchas

RE can get quite complicated and cryptic a lot of the times. But sometimes, if something is not working as expected, it could be because of quirky corner cases.

Some RE engines match character literally if an escape sequence is not defined. Python raises an exception for such cases. Apart from sequences defined for RE, these are allowed: `\a \b \f \n \r \t \u \U \v \x \\` where `\b` means backspace only in character classes and `\u \U` are valid only in Unicode patterns.

```
>>> bool(re.search(r'\t', 'cat\tdog'))
True
>>> bool(re.search(r'\c', 'cat\tdog'))
re.error: bad escape \c at position 0
```

There is an additional start/end of line match after last newline character if line anchors are used as standalone pattern. End of line match after newline is straightforward to understand as `$` matches both end of line and end of string.

```
# note also the use of special group for enabling multiline flag
>>> print(re.sub(r'(?m)^', r'foo ', '1\n2\n'))
foo 1
foo 2
foo

>>> print(re.sub(r'(?m)$', r' baz', '1\n2\n'))
1 baz
2 baz
 baz
```

How much does `*` or `*+` match?

```
# there is an extra empty string match at end of non-empty columns
>>> re.sub(r'[^,]*', r'{\g<0>}', ',cat,tiger')
'{},{cat}{},{tiger}{}'
>>> regex.sub(r'[^,]*+', r'{\g<0>}', ',cat,tiger')
'{},{cat}{},{tiger}{}'

# use lookarounds as a workaround
>>> re.sub(r'(?<![^,])[^,]*', r'{\g<0>}', ',cat,tiger')
'{},{cat},{tiger}'
```

Referring to text matched by a capture group with a quantifier will give only the last match, not entire match. Use a non-capturing group inside a capture group to get the entire matched portion.

```
>>> re.sub(r'\A([^,]+,){3}([^,]+)', r'\1(\2)', '1,2,3,4,5,6,7', count=1)
'3,(4),5,6,7'
>>> re.sub(r'\A((?:[^,]+,){3})([^,]+)', r'\1(\2)', '1,2,3,4,5,6,7', count=1)
'1,2,3,(4),5,6,7'

# as mentioned earlier, findall can be useful for debugging purposes
>>> re.findall(r'([^,]+,){3}', '1,2,3,4,5,6,7')
['3,', '6,']
>>> re.findall(r'(?:[^,]+,){3}', '1,2,3,4,5,6,7')
['1,2,3,', '4,5,6,']
```

When using `flags` options with `regex` module, the constants should also be used from `regex` module. A typical workflow shown below:

```
# Using re module, unsure if a feature is available
>>> re.findall(r'[[:word:]]+', 'fox:αλεπού,eagle:αετός', flags=re.A)
__main__:1: FutureWarning: Possible nested set at position 1
[]
# Ok, convert re to regex
# Oops, output is still wrong
>>> regex.findall(r'[[:word:]]+', 'fox:αλεπού,eagle:αετός', flags=re.A)
```

```
['fox', 'αλεπού', 'eagle', 'αετός']

# Finally correct solution, the constant had to be changed as well
>>> regex.findall(r'[[:word:]]+', 'fox:αλεπού,eagle:αετός', flags=regex.A)
['fox', 'eagle']
```

Speaking of `flags`, try to always use it as keyword argument. Using it as positional argument leads to a common mistake between `re.findall` and `re.sub` due to difference in placement. Their syntax, as per the docs, is shown below:

```
re.findall(pattern, string, flags=0)

re.sub(pattern, repl, string, count=0, flags=0)
```

Hope you have found Python regular expressions an interesting topic to learn. Sooner or later, you'll need to use them if you are facing plenty of text processing tasks. At the same time, knowing when to use normal string methods and knowing when to reach for other text parsing modules is important. Happy coding!

Further Reading

Note that most of these resources are not specific to Python, so use them with caution and check if they apply to Python's syntax and features

- docs.python: Regular Expression HOWTO
- stackoverflow: python regex
- CommonRegex - collection of common regular expressions
- Generate strings that match a given regular expression
- stackoverflow: regex FAQ
  - stackoverflow: regex tag is a good source of exercise questions
- rexegg - tutorials, tricks and more
- regular-expressions - tutorials and tools
- regexcrossword - tutorials and puzzles
- regexper - for visualization
- regex101 - visual aid and online testing tool for regular expressions, select flavor as Python before use
- swtch - stuff about regular expression implementation engines

Here's some links for specific topics:

- rexegg: best regex trick
- regular-expressions: matching numeric ranges
- regular-expressions: Continuing at The End of The Previous Match
- regular-expressions: Zero-Length Matches
- stackoverflow: Greedy vs Reluctant vs Possessive Quantifiers
- stackoverflow: named captures as a dict
- stackoverflow: Is it worth using re.compile?