

Groupings and backreferences

This chapter will show how to reuse portion matched by capture groups via backreferences within RE definition and replacement section. You'll also learn some of the special grouping syntax for cases where plain capture groups isn't enough.

Backreference

Backreferences are like variables in a programming language. You have already seen how to use `re.Match` object to refer to the text captured by groups. Backreferences provide the same functionality, with the advantage that these can be directly used in RE definition as well as replacement section without having to invoke `re.Match` objects. Another advantage is that you can apply quantifiers to backreferences.

The syntax is `\N` or `\g<N>` where `N` is the capture group you want. The below syntax variations is applicable for **replacement section**, assuming they are used within raw strings.

- `\1`, `\2` up to `\99` to refer to the corresponding capture group
 - provided there are no digit characters after
 - `\0` and `\NNN` will be interpreted as octal value
- `\g<1>`, `\g<2>` etc (not limited to 99) to refer to the corresponding capture group
 - this also helps to avoid ambiguity between backreference and digits that follow
- `\g<0>` to refer to entire matched portion, similar to index `0` of `re.Match` objects
 - `\0` cannot be used because numbers starting with `0` are treated as octal value

Here's some examples with `\N` syntax.

```
# remove square brackets that surround digit characters
# note that use of raw strings for replacement string
>>> 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'

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

Here's some examples with `\g<N>` syntax.

```
# ambiguity between \N and digit characters part of replacement string
>>> re.sub(r'\[(\d+)\]', r'(\15)', '[52] apples and [31] mangoes')
re.error: invalid group reference 15 at position 2
# \g<N> solves this issue
>>> re.sub(r'\[(\d+)\]', r'\g<1>5)', '[52] apples and [31] mangoes')
'(525) apples and (315) mangoes'
# you can also use octal escapes
>>> re.sub(r'\[(\d+)\]', r'\1\065)', '[52] apples and [31] mangoes')
'(525) apples and (315) mangoes'

# add something around the matched strings using \g<0>
>>> re.sub(r'[a-z]+', r'\g<0>}', '[52] apples and [31] mangoes')
'[52] {apples} {and} [31] {mangoes}'

# note the use of '+' instead of '*' quantifier to avoid empty matching
```


```
>>> re.sub(r'.+', r'Hi. \g<0>. Have a nice day', 'Hello world')
'Hi. Hello world. Have a nice day'

# duplicate first field and add it as last field
>>> re.sub(r'\A([^\s]+),.+', r'\g<0>,\1', 'fork,42,nice,3.14')
'fork,42,nice,3.14,fork'
```

Here's some examples for using backreferences within **RE definition**. Only `\N` syntax is available for use.

```
# 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
# note the use of quantifier on backreferences
# use \W+ instead of space to cover cases like 'a;a<-;a'
>>> re.sub(r'\b(\w+)(\1)+\b', r'\1', 'aa a a a 42 f_1 f_1 f_13.14')
'aa a 42 f_1 f_13.14'
```

 Since `\g<N>` syntax is not available in RE definition, use formats like hexadecimal escapes to avoid ambiguity between normal digit characters and backreferences.

```
>>> s = 'abcdefghijklmna1d'

# even though there's only one capture group, \11 will give an error
>>> re.sub(r'(.)*\11', 'X', s)
re.error: invalid group reference 11 at position 6
# use escapes for the digit portion to distinguish from the backreference
>>> re.sub(r'(.)*\1\x31', 'X', s)
'Xd'

# there are 12 capture groups here, so no error
# but requirement is \1 as backreference and 1 as normal digit
>>> re.sub(r'(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)*\11', 'X', s)
'abcdefghijklmna1d'
# use escapes again
>>> re.sub(r'(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)(.)*\1\x31', 'X', s)
'Xd'
```

 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 `(\d[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. `regex` module supports [Subexpression calls](#) to refer to the RE itself.

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` as seen in [Working with matched portions](#) chapter.

When backreferencing is not required, you can use a non-capturing group to avoid undesired behavior. It also helps to avoid keeping a track of capture group numbers when that particular group is not needed for backreferencing. The syntax is `(?:pat)` to define a non-capturing group. You'll see many more of such special groups starting with `(?` syntax 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(?:[^\d,]+){3}([^\d,]+)', r'\1(\3)', '1,2,3,4,5,6,7')
'1,2,3,(4),5,6,7'
# using non-capturing groups, only relevant groups have to be tracked
>>> re.sub(r'\A(?:[^\d,]+){3}([^\d,]+)', r'\1(\2)', '1,2,3,4,5,6,7')
'1,2,3,(4),5,6,7'
```

Referring to text matched by a capture group with a quantifier will give only the last match, not entire match. Use a capture group around the grouping and quantifier together to get the entire matching portion. In such cases, the inner grouping is an ideal candidate to use non-capturing group.

```
>>> s = 'hi 123123123 bye 456123456'
>>> re.findall(r'(123)+', s)
['123', '123']
>>> re.findall(r'(?:123)+', s)
['123123123', '123']
# note that this issue doesn't affect substitutions
```

```
>>> row = 'one,2,3.14,42,five'
# surround only fourth column with double quotes
# note the loss of columns in the first case
>>> re.sub(r'\A(?:[^\d,]+){3}([^\d,]+)', r'\1"\2"', row)
'3.14,"42",five'
>>> re.sub(r'\A(?:[^\d,]+){3}([^\d,]+)', r'\1"\2"', row)
'one,2,3.14,"42",five'
```

However, there are situations where capture groups cannot be avoided. In such cases, you'd need to manually work with `re.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 to the rescue
>>> 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>pat)` for naming the capture groups. The name used should be a valid Python identifier. Use `'name'` for `re.Match` objects, `\g<name>` in replacement section and `(?P=name)` for backreferencing in RE definition. 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>', 'good,bad 42,24')
'bad,good 24,42'
```

```

>>> s = 'aa a a a 42 f_1 f_1 f_13.14'
>>> re.sub(r'\b(?:P<dup>\w+)(?:P=dup))+\b', r'\g<dup>', s)
'aa a 42 f_1 f_13.14'

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

```

You can use `groupdict` method on the `re.Match` object to extract the portions matched by named capture groups as a `dict` object. The capture group name will be the **key** and the portion matched by the group will be the **value**.

```

# single match
>>> details = '2018-10-25,car,2346'
>>> re.search(r'(?P<date>[^,]+),(?P<product>[^,]+)', details).groupdict()
{'date': '2018-10-25', 'product': 'car'}

# normal groups won't be part of the output
>>> re.search(r'(?P<date>[^,]+),([^,]+)', details).groupdict()
{'date': '2018-10-25'}

# multiple matches
>>> s = 'good,bad 42,24'
>>> [m.groupdict() for m in re.finditer(r'(?P<fw>\w+),(?P<sw>\w+)', s)]
[{'fw': 'good', 'sw': 'bad'}, {'fw': '42', 'sw': '24'}]

```



Conditional groups



This special grouping allows you to add a condition that depends on whether a capture group succeeded in matching. You can also add an optional else condition. The syntax as per the docs is shown below.

```
(?(id/name)yes-pattern|no-pattern)
```

Here `id` means the `N` used to backreference a capture group and `name` refers to the identifier used for a named capture group. Here's an example with **yes-pattern** alone being used. The task is to match elements containing word characters only or if it additionally starts with a double quote, it must end with a double quote.

```

>>> words = ['"hi"', 'bye', 'bad"', '"good"', '42', '"3']
>>> pat = re.compile(r'(")?\w+(?(1)"')
>>> [w for w in words if pat.fullmatch(w)]
['"hi"', 'bye', '"good"', '42']

# for this simple case, you can also expand it manually
# but for complex patterns, it is better to use conditional groups
# as it will avoid repeating the complex pattern
>>> [w for w in words if re.fullmatch(r'"'\w+"|\w+', w)]
['"hi"', 'bye', '"good"', '42']

# cannot simply use ? quantifier as they are independent, not constrained
>>> [w for w in words if re.fullmatch(r'""?\w+"?', w)]
['"hi"', 'bye', 'bad"', '"good"', '42', '"3']
# also, fullmatch plays a big role in avoiding partial match
>>> [w for w in words if pat.search(w)]
['"hi"', 'bye', 'bad"', '"good"', '42', '"3']

```



Here's an example with **no-pattern** as well.

```
# filter elements containing word characters surrounded by ()
# or, containing word characters separated by a hyphen
>>> words = ['(hi)', 'good-bye', 'bad', '(42)', '-oh', 'i-j', '(-)']

# same as: r'\(\w+\)|\w+~\w+'
>>> pat = re.compile(r'\(\)?\w+(?(1)\)|~\w+)')
>>> [w for w in words if pat.fullmatch(w)]
['(hi)', 'good-bye', '(42)', 'i-j']
```



Conditional groups have a very specific use case, and it is generally helpful for those cases. The main advantage is that it prevents pattern duplication, although that can also be achieved using [Subexpression calls](#) with `regex` module. Another advantage is that if the common pattern uses capture groups, then the duplication alternation method will need different backreference numbers.

Match.expand

The `expand` method on `re.Match` objects accepts syntax similar to the replacement section of `re.sub` function. The difference is that `expand` method returns only the string after backreference expansion, instead of entire input string with modified content.

```
# re.sub vs Match.expand
>>> re.sub(r'w(.*)m', r'[\1]', 'awesome')
'a[eso]e'
>>> re.search(r'w(.*)m', 'awesome').expand(r'[\1]')
'[eso]'
```

```
# example with re.finditer
>>> dates = '2020/04/25,1986/03/02,77/12/31'
>>> m_iter = re.finditer(r'([^\s/]+)/([^\s/]+)/[^\s,]+?', dates)
# same as: [f'Month:{m[2]}, Year:{m[1]}' for m in m_iter]
>>> [m.expand(r'Month:\2, Year:\1') for m in m_iter]
['Month:04, Year:2020', 'Month:03, Year:1986', 'Month:12, Year:77']
```



Cheatsheet and Summary

Note	Description
<code>\N</code>	backreference, gives matched portion of Nth capture group
	applies to both RE definition and replacement section
	possible values: <code>\1</code> , <code>\2</code> up to <code>\99</code> provided no more digits
	<code>\0</code> and <code>\NNN</code> will be treated as octal escapes
<code>\g<N></code>	backreference, gives matched portion of Nth capture group
	applies only to replacement section
	use escapes to prevent ambiguity in RE definition
	possible values: <code>\g<0></code> , <code>\g<1></code> , etc (not limited to 99)
	<code>\g<0></code> refers to entire matched portion
<code>(?:pat)</code>	non-capturing group
	useful wherever grouping is required, but not backreference
<code>(?P<name>pat)</code>	named capture group
	refer as 'name' in <code>re.Match</code> object
	refer as <code>(?P=name)</code> in RE definition
	refer as <code>\g<name></code> in replacement section
	can also use <code>\N</code> and <code>\g<N></code> format if needed
<code>groupdict</code>	method applied on a <code>re.Match</code> object

Note	Description
	gives named capture group portions as a <code>dict</code>
<code>(?(id/name)yes no)</code>	conditional group
	match <code>yes-pattern</code> if backreferenced group succeeded
	else, match <code>no-pattern</code> which is optional
<code>expand</code>	method applied on a <code>re.Match</code> object
	accepts syntax like replacement section of <code>re.sub</code>
	gives back only string after backreference expansion

This chapter showed how to use backreferences to refer to portion matched by capture groups in both RE definition and replacement section. When capture groups results in unwanted behavior change (ex: `re.findall` and `re.split`), you can use non-capturing groups instead. Named capture groups add clarity to patterns and you can use `groupdict` method on a `re.Match` object to get a `dict` of matched portions. Conditional groups allows you to take an action based on another capture group succeeding or failing to match. There are more special groups to be discussed in coming chapters.

Exercises

a) Replace the space character that occurs after a word ending with `a` or `r` with a newline character.

```
>>> ip = 'area not a _a2_ roar took 22'

>>> print(re.sub())      ##### add your solution here
area
not a
_a2_ roar
took 22
```

b) Add `[]` around words starting with `s` and containing `e` and `t` in any order.

```
>>> ip = 'sequoia subtle exhibit asset sets tests site'

##### add your solution here
'sequoia [subtle] exhibit asset [sets] tests [site]'
```

c) Replace all whole words with `x` that start and end with the same word character. Single character word should get replaced with `x` too, as it satisfies the stated condition.

```
>>> ip = 'oreo not a _a2_ roar took 22'

##### add your solution here
'X not X X X took X'
```

d) 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'
```

e) 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)'
```

f) Count the number of whole words that have at least two occurrences of consecutive repeated alphabets. For example, words like `stillness` and `Committee` should be counted but not words like `root` or `readable` or `rotational`.

```
>>> ip = '''oppressed abandon accommodation bloodless
... carelessness committed apparition innkeeper
... occasionally afforded embarrassment foolishness
... depended successfully succeeded
... possession cleanliness suppress'''

##### add your solution here
13
```

g) For the given input string, replace all occurrences of digit sequences with only the unique non-repeating sequence. For example, `232323` should be changed to `23` and `897897` should be changed to `897`. If there no repeats (for example `1234`) or if the repeats end prematurely (for example `12121`), it should not be changed.

```
>>> ip = '1234 2323 453545354535 9339 11 60260260'

##### add your solution here
'1234 23 4535 9339 1 60260260'
```

h) Replace sequences made up of words separated by `:` or `.` by the first word of the sequence. Such sequences will end when `:` or `.` is not followed by a word character.

```
>>> ip = 'wow:Good:2_two:five: hi-2 bye kite.777.water.'

##### add your solution here
'wow hi-2 bye kite'
```

i) Replace sequences made up of words separated by `:` or `.` by the last word of the sequence. Such sequences will end when `:` or `.` is not followed by a word character.

```
>>> ip = 'wow:Good:2_two:five: hi-2 bye kite.777.water.'

##### add your solution here
'five hi-2 bye water'
```

j) Split the given input string on one or more repeated sequence of `cat`.

```
>>> ip = 'firecatlioncatcatcatbearcatcatparrot'

##### add your solution here
['fire', 'lion', 'bear', 'parrot']
```

k) For the given input string, find all occurrences of digit sequences with at least one repeating sequence. For example, `232323` and `897897`. If the repeats end prematurely, for example `12121`, it should not be matched.

```
>>> ip = '1234 2323 453545354535 9339 11 60260260'

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

# entire sequences in the output
##### add your solution here
['2323', '453545354535', '11']
```

```
# only the unique sequence in the output
##### add your solution here
['23', '4535', '1']
```

l) Convert the comma separated strings to corresponding dict objects as shown below. The keys are name, maths and phy for the three fields in the input strings.

```
>>> row1 = 'rohan,75,89'
>>> row2 = 'rose,88,92'

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

##### add your solution here for row1
{'name': 'rohan', 'maths': '75', 'phy': '89'}
##### add your solution here for row2
{'name': 'rose', 'maths': '88', 'phy': '92'}
```

m) Surround all whole words with (). Additionally, if the whole word is imp or ant, delete them. Can you do it with single substitution?

```
>>> ip = 'tiger imp goat eagle ant important'

##### add your solution here
'(tiger) () (goat) (eagle) () (important)'
```

n) Filter all elements that contains a sequence of lowercase alphabets followed by - followed by digits. They can be optionally surrounded by {} and {}. Any partial match shouldn't be part of the output.

```
>>> ip = ['{{apple-150}}', '{{mango2-100}}', '{{cherry-200}', 'grape-87']

##### add your solution here
['{{apple-150}}', 'grape-87']
```

o) The given input string has sequences made up of words separated by : or . and such sequences will end when : or . is not followed by a word character. For all such sequences, display only the last word followed by - followed by first word.

```
>>> ip = 'wow:Good:2_two:five: hi-2 bye kite.777.water.'

##### add your solution here
['five-wow', 'water-kite']
```