Python 3: Regular Expressions

Bob Dowling rjd4@cam.ac.uk

29 October 2012

Prerequisites

This self-paced course assumes that you have a knowledge of Python 3 equivalent to having completed one or other of

- Python 3: Introduction for Absolute Beginners, or
- Python 3: Introduction for Those with Programming Experience

Some experience beyond these courses is always useful but no other course is assumed.

The course also assumes that you know how to use a Unix text editor (`gedit`, `emacs`, `vi`, …).

Facilities for this session

The computers in this room have been prepared for these self-paced courses. They are already logged in with course IDs and have home directories specially prepared. Please do not log in under any other ID.

At the end of this session the home directories will be cleared. Any files you leave in them will be deleted. Please copy any files you want to keep.

The home directories contain a number of subdirectories one for each topic.

For this topic please enter directory `regexp`. All work will be completed there:

```
$ cd regexp
$ pwd
/home/y550/regexp
$
```

These courses are held in a room with two demonstrators. If you get stuck or confused, or if you just have a question raised by something you read, please ask!

These handouts and the prepared folders to go with them can be downloaded from
`www.ucs.cam.ac.uk/docs/course-notes/unix-courses/pythontopics`

The formal Python 3 documentation for the topics covered here can be found online at
`docs.python.org/release/3.2.3/library/re.html`

Table of Contents

Notation

Warnings

Warnings are marked like this. These sections are used to highlight common mistakes or misconceptions.

Exercises

Exercise 0

Exercises are marked like this. You are expected to complete all exercises. Some of them do depend on previous exercises being successfully completed.

Input and output

Material appearing in a terminal is presented like this:

```
$ more lorem.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
--More--(44%)
```

The material you type is presented like this: **`ls`**. (Bold face, typewriter font.)

The material the computer responds with is presented like this: “`Lorem ipsum`”. (Typewriter font again but in a normal face.)

Keys on the keyboard

Keys on the keyboard will be shown as the symbol on the keyboard surrounded by square brackets, so the “A key” will be written “[A]”. Note that the return key (pressed at the end of every line of commands) is written “[↲]”, the shift key as “[⇧]”, and the tab key as “[↹]”. Pressing more than one key at the same time (such as pressing the shift key down while pressing the A key) will be written as “[⇧]+[A]”. Note that pressing [A] generates the lower case letter “a”. To get the upper case letter “A” you need to press [⇧]+[A].

Content of files

The content[1] of files (with a comment) will be shown like this:

```
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat nulla pariatur.                                  This is a comment about the line.
```

1 The example text here is the famous “lorem ipsum” dummy text used in the printing and typesetting industry. It dates back to the 1500s. See http://www.lipsum.com/ for more information.

What's in this course

1. What is a regular expression?
2. Case (in)sensitivity
3. A more complex example
4. Character choices
5. Counting in regular expressions
6. "Escaping" characters
7. Alternation
8. "Verbose" regular expressions
9. Components of matching text
10. Named groups
11. Ambiguity and greed
12. Internal references

What is a regular expression?

A regular expression is simply some means to write down a pattern describing some text. (There is a formal mathematical definition but we're not bothering with that here. What the computing world calls regular expressions and what the strict mathematical grammarians call regular expressions are slightly different things.)

For example we might like to say "a series of digits" or "a single lower case letter followed by some digits". There are terms in regular expression language for all of these concepts. The language of these patterns can seem intimidating at first:

"a series of digits"	`\d+`
"a lower case letter followed by some digits"	`[a-z]\d+`
"a mixture of characters except for new line, followed by a full stop and one or more letters or numbers"	`.+\w+`

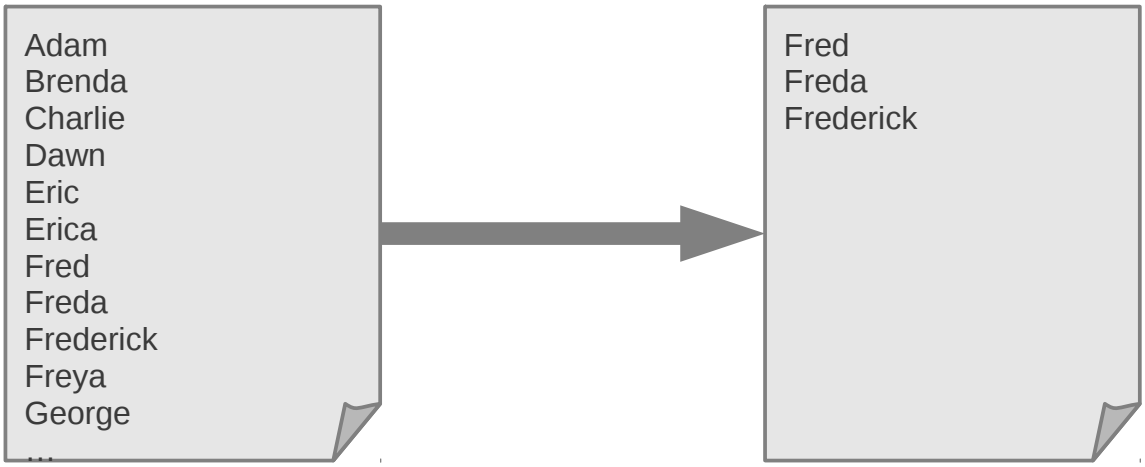
We will cover what these mean very soon. We will start with a "trivial" regular expression, however, which simply matches a fixed bit of text.

We will set ourselves the goal of writing a "filter script" that reads lines in from the "standard input" and writes its output to "standard output". What this means in practice is that if we want to identify both an input and output file we run our scripts like this:

```
$ python3 example01.py < input_file > output_file
```

If we omit the "`< input_file`" our input comes from the keyboard. If we omit the "`> output_file`" our output goes to the screen.

Our initial target will be to find the lines in the file `names.txt` which include the text "`Fred`":



We will start building up our script with the non-regular expression elements:

```
import regular expression module
import sys                                  # For the I/O

define pattern
set up regular expression

for line in sys.stdin:                      # Read one line at a time
    compare line to regular expression
    if the line matches:
        sys.stdout.write(line)              # Write out matching lines
```

Now we have to add some Python for the regular expression handling.

The regular expression module in Python is called, simply, "`re`", so our first line becomes

```
import re
```

Patterns are simply text strings. We will cover the syntax of regular expressions very shortly but for the time being, we will look for the text "Fred" in files. The regular expression for this is simply the text itself, "Fred", so the line that defines the pattern becomes

```
pattern = 'Fred'
```

Now we have to turn that simple text into a Python object that can actually *do* something. We are going to take the text "Fred" and we are going to convert it into a Python regular expression object. We call this process "compilation" and in a very real sense the plain text "Fred" is the source code for the mini-program we are about to create:

```
regexp = re.compile(pattern)
```

The `regexp` object is the Python object that will actually go looking for matching lines. So now we need to know how to do the comparison of a trial line of text against the pattern we are looking for. The `regexp` object has a method that does precisely this: `search()`. This method takes the text being queried as its argument and looks for the pattern in it. The method returns a "match object" that describes if and how the text matches the pattern:

```
match = regexp.search(line)
```

This match object can be converted to a boolean. If there are no matches, so the pattern does not appear in

the text, then it converts to `False`. If there is a match then it converts to `True`. As a result of this we can test for a matching line very easily:

```
if match:
```

The following block of code will only be run if there is a match.

Putting this together gives us our first regular expression Python script that searches for “Fred”:

```
import re
import sys
pattern = 'Fred'
regexp = re.compile(pattern)
for line in sys.stdin:
    match = regexp.search(line)
    if match:
        sys.stdout.write(line)
```

We can run this script, `example01.py`, to check it works:

```
$ python3 example01.py < names.txt
Fred
Freda
Frederic
$
```

You should make sure that you understand this script before proceeding.

Exercise 1

Write a script, `exercise01.py`, from scratch that looks for the string “`bert`” in the file of names.

Case insensitivity

Our example script pulled Fred, Freda, and Frederick from the list of names, but omitted Manfred. The exercise (if completed correctly) should have pulled out Norbert, Robert, and Roberta but not Bertram. Python regular expressions are case sensitive, like everything else in Python. The text “Fred” is not the same as the text “fred”.

We can build ourselves a case insensitive regular expression mini-program if we want to. The `re.compile()` function we saw earlier can take a second, optional argument. This argument is a set of flags which modify how the regular expression works. One of these flags makes it case insensitive.

The options are set as a series of values that need to be added together. We’re currently only interested in one of them, though, so we can give “`re.IGNORECASE`” (the `IGNORECASE` constant from the `re` module) as the second argument.

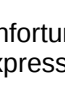
For those of you who dislike long options, the `re.I` constant is a synonym for the `re.IGNORECASE` constant. We encourage the long forms as better reminders of what the options mean for when you come back to this script having not looked at it for six months:

```
regexp = re.compile(pattern, re.IGNORECASE)
```

Exercise 2

Copy your script `exercise01.py` to `exercise02.py` and edit `exercise02.py` to be case insensitive. Run it again.

A more complex example



Unfortunately, there's a fair bit of reading to do at this point to get you into regular expressions. Take a deep breath and get ready for the next four pages.

Searching for fixed text such as "`Fred`" or "`bert`" is a valid task, but regular expressions are capable of much more. We will now extend the `example01.py` script to do some serous analysis of an input file.

The file "`atoms.log`" is the output of a set of programs which do something involving atoms of the elements. (It's a fictitious example, so don't obsess on the detail.) It has a collection of lines corresponding to how various runs of a program completed.

Some are simple success lines such as the first line:

```
RUN 000001 COMPLETED.  OUTPUT IN FILE hydrogen.dat.
```

Others have additional information indicating that things did not go so well:

```
RUN 000039 COMPLETED.  OUTPUT IN FILE yttrium.dat. 1 UNDERFLOW WARNING.
RUN 000057 COMPLETED.  OUTPUT IN FILE lanthanum.dat. ALGORITHM DID NOT CONVERGE
AFTER 100000 ITERATIONS.
RUN 000064 COMPLETED.  OUTPUT IN FILE gadolinium.dat. OVERFLOW ERROR.
```

Our job will be to unpick the "good lines" from the rest.

We will build the pattern required for these good lines bit by bit. It helps to have some lines "in mind" while developing the pattern, and to consider which bits change between lines and which bits don't.

Because we are going to be using some leading and trailing spaces in our strings we will mark them explicitly in the examples:

```
RUN␣000001␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣hydrogen.dat.
```

```
RUN␣000017␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣chlorine.dat.
```

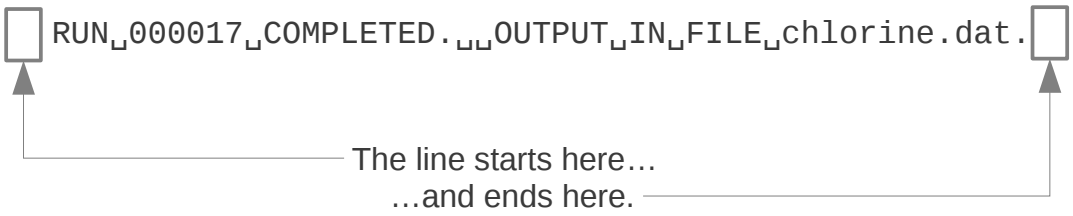
The fixed text is shown above. Note that while the element part of the file name varies, its suffix is constant.

```
RUN␣000017␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣chlorine.dat.
```

six digits

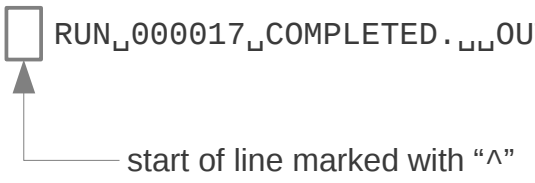
sequence of lower-case letters

The first part of the line that varies is the set of six digits. Note that we are lucky that it is always six digits. More realistic output might have varying numbers of digits: 2 digits for "17" as in the slide but only one digit for "9". The second varying part is the primary part of the file name.



What we have described to date matches all the lines. They all start with that same sentence. What distinguishes the good lines from the bad is that this is all there is. The lines start and stop with exactly this, no more and no less.

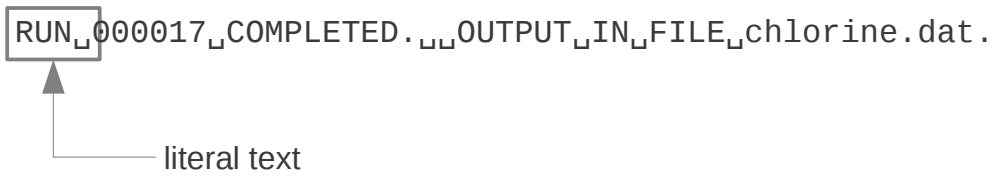
It is good practice to match against as much of the line as possible as it lessens the chance of accidentally matching a line you didn't plan to. Later on it will become essential as we will be extracting elements from the line.



We will be building the pattern at the bottom of the slide. We start by saying that the line begins here. Nothing may precede it.

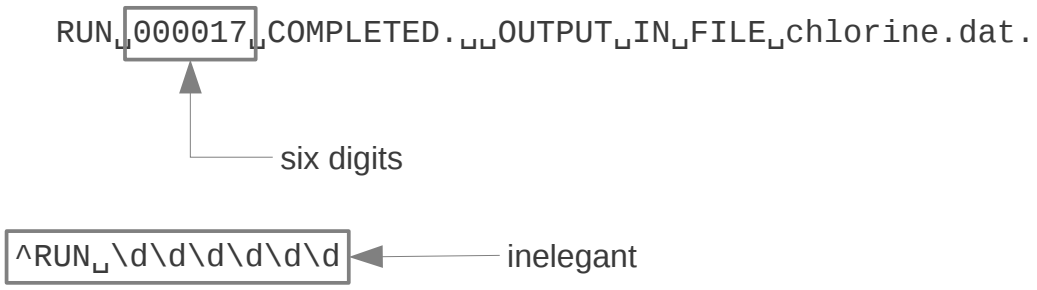
The start of line is represented with the "caret" or "circumflex" character, "^". This is the first of the special regular expressions that mean something other than just the literal text.

The "^" is known as an anchor, because it forces the pattern to match only at a fixed point (the start, in this case) of the line. Such patterns are called anchored patterns. Patterns which don't have any anchors in them are known as (surprise!) unanchored patterns.



Next comes some literal text. We just add this to the pattern as is.

There's one gotcha we will return to later. It's easy to get the wrong number of spaces or to mistake a tab stop for a space. In this example it's a single space, but we will learn how to cope with generic "white space" later.



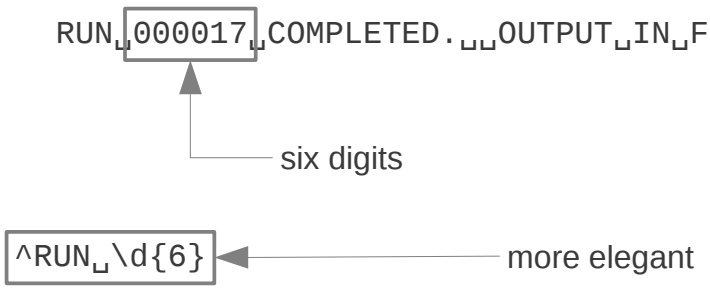
Next comes a run of six digits. There are two approaches we can take here. A digit can be regarded as a character between “0” and “9” in the character set used, but it is more elegant to have a pattern that explicitly says “a digit”.

The sequence “`[0-9]`” has the meaning “one character between “0” and “9” in the character set. (We will meet this use of square brackets in detail in a few slides’ time.) The sequence “`\d`” means exactly “one digit”.

`[0-9]` “any single character between 0 and 9”

`\d` “any digit”

However, a line of six instances of “`\d`” is not particularly elegant. Can you imagine counting them if there were sixty rather than six?

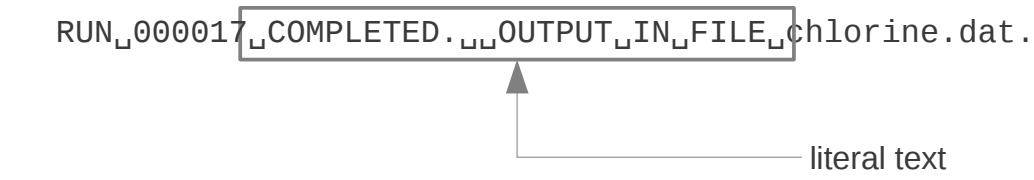


Regular expression pattern language has a solution to this inelegance. Following any pattern with a number in curly brackets (“braces”) means to iterate that pattern that many times.

Note that “`\d{6}`” means “six digits in a row”. It does not mean “the same digit six times”. We will see how to describe that later.

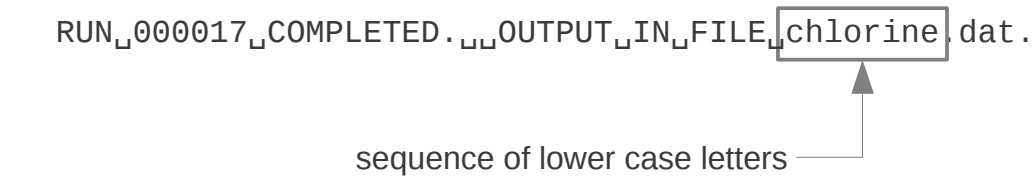
The syntax can be extended:

`\d{6}`	six digits
`\d{5,7}`	five, six or seven digits
`\d{5,}`	five or more digits
`\d{,7}`	no more than seven digits (including no digits)




```
^RUN␣\d{6}␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣
```

Next comes some more fixed text. As ever, don't forget the leading, trailing and double spaces.




```
^RUN␣\d{6}␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣[a-z]+
```

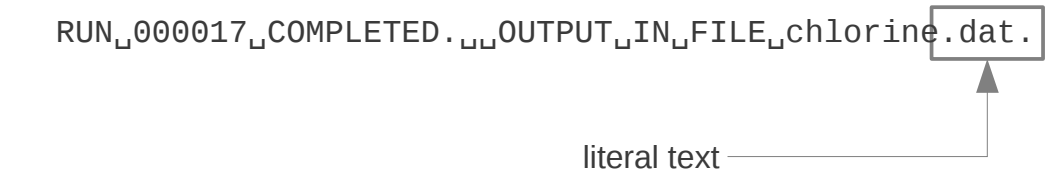
Next comes the name of the element. We will ignore for these purposes the fact that we know these are the names of elements. For our purposes they are sequences of lower case letters.

This time we will use the square bracket notation. This is identical to the wild cards used by Unix shells, if you are already familiar with that syntax. The regular expression pattern "`[aeiou]`" means "exactly one character which can be either an 'a', an 'e', an '`i`', an 'o', or a 'u'".

The slight variant "`[a-m]`" means "exactly one character between 'a' and '`m`' inclusive in the character set". In the standard computing character sets (with no internationalisation turned on) the digits, the lower case letters, and the upper case letters form uninterrupted runs. So "[0-9]" will match a single digit. "`[a-z]`" will match a single lower case letter. "`[A-Z]`" will match a single upper case letter.

But we don't want to match a *single* lower case letter. We want to match an unknown number of them.

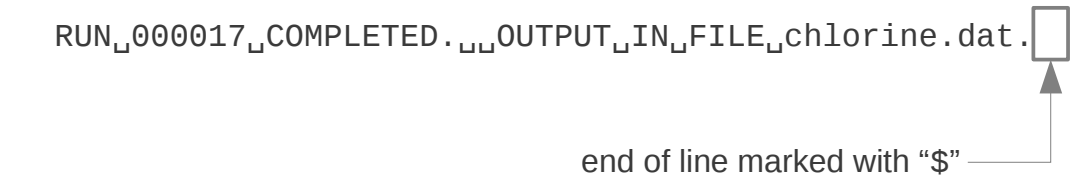
Any pattern can be followed by a "+" to mean "repeat the pattern one or more times". So "`[a-z]+`" matches a sequence of one or more lower case letters. (Again, it does not mean "the same lower case letter multiple times".) It is equivalent to "`[a-z]{1,}`".




```
^RUN␣\d{6}␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣[a-z]+.dat.
```

Next we have the closing literal text. (Strictly speaking the dot is a special character in regular expressions but we will address that in a later slide.)

```
RUN␣000017␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣chlorine.dat.␣
```




```
^RUN␣\d{6}␣COMPLETED.␣␣OUTPUT␣IN␣FILE␣[a-z]+.dat.$
```

Finally, and crucially, we identify this as the end of the line. The lines with warnings and errors go beyond this point.

The dollar character, “$”, marks the end of the line. This is another anchor, since it forces the pattern to match only at another fixed place (the end) of the line.

Note that although we are using both ^ and $ in our pattern, you don’t have to always use both of them in a pattern. You may use both, or only one, or neither, depending on what you are trying to match.

Exercise 3

Copy your script `exercise01.py` to `exercise03.py` and edit `exercise03.py` to use this extended regular expression for processing the file `atoms.log`.
Test that it works:

```
$ python3 exercise03.py < atoms.log
```

Special codes for regular expressions

We have seen a few special codes in our previous example that mean something special in regular expressions. :

`\A`	`^`	Anchor start of line
`\Z`	`$`	Anchor end of line
`\d`	`[0-9]`	Any digit
`\D`		Any non-digit
	`[abc]`	Any one of “a”, “b“, or “c”

We will build up this list as we proceed. Note that there are alternative forms of various codes.

First, we will look more closely at what can go in square brackets.

If we have just a set of simple characters (e.g. “`[aeiou]`”) then it matches any one character from that set. Note that the set of simple characters can include a space, e.g. “`[␣aeiou]`” matches a space or an “a” or an “e” or an “i” or an “o” or a “u”.

If we put a dash between two characters then it means any one character from that range. So “`[a-z]`” is exactly equivalent to “`[abcdefghijklmnopqrstuvwxyz]`”. We can repeat this for multiple ranges, so “`[A-Za-z]`” is equivalent to “`[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]`”.

If we want one of the characters in the set to be a dash, “`-`”, there are two ways we can do this. We can precede the dash with a backspace “`\-`” to mean “include the character ‘-’ in the set of characters we want to match”, e.g. “`[A-Za-z\-]`” means “match any alphabetic character or a dash”. Alternatively, we can make the first character in the set a dash in which case it will be interpreted as a literal dash (“`-`”) rather than indicating a range of characters, e.g. “`[-A-za-z]`” also means “match any alphabetic character or a dash”.

The backslash approach is more generally applicable and can also be used to include a close square bracket in the set of characters.

`[aeiou]`	Any one of “a”, “e”, “i”, “o”, or “u”.
`[A-Z]`	Any upper case letter
`[A-Za-z]`	Any upper or lower case letter
`[A-Z\]]`	Any upper case letter or a square bracket
`[A-Z\-]`	Any upper case letter or a dash

There is one last piece of regular expression syntax for square brackets. In addition to specifying “any one of these characters” we also need to be able to specify “any one character that is *not* one of these”. This is done by immediately following the open square bracket with a circumflex character, “^”. Note that this is a second, different use of the circumflex. We have already seen it used to anchor a pattern to the start of the line. It can’t mean this inside square brackets so there is no possibility for confusion.

`[^aeiou]`	Any one character that is not a lower case vowel
`[^A-Z]`	Any one character that is not an upper case letter
`[^A-Za-z]`	Any one character that is not an upper or lower case letter
`[^A-Z\]]`	Any one character that is not an upper case letter or a square bracket
`[^A-Z\-]`	Any one character that is not an upper case letter or a dash

So, for example, the regular expression “`^[Aa][Aa]`” will match any line that begins with two A’s (in either case) and the expression “`[^Aa][Aa]`” will match any line that contains a “not-a” followed by an “a” (again, in either case).

Exercise 4

There are three separate parts to this exercise.
The file `/usr/share/dict/words` contains every word recognised by a simple spell checker.
Edit the `pattern=...` line (and no others) in `exercise04.py` to find every word that matches
(a) “`^[Aa][Aa]`”
(b) “`[^Aa][Aa]`”
(c) “lines that have a “double-a” in them but which don’t start with an “a”

```
$ python3 exercise04.py
```

(The script opens the correct file automatically.)

We also saw that we can count in regular expressions. These counting modifiers appear below after the example pattern “`[abc]`”. They can follow *any* regular expression pattern.

`[abc]`	Any one of ‘a’, ‘b’ or ‘c’.
`[abc]+`	One or more ‘a’, ‘b’ or ‘c’.
`[abc]?`	Zero or one ‘a’, ‘b’ or ‘c’.
`[abc]*`	Zero or more ‘a’, ‘b’ or ‘c’.
`[abc]{6}`	Exactly 6 of ‘a’, ‘b’ or ‘c’.
`[abc]{5,7}`	5, 6 or 7 of ‘a’, ‘b’ or ‘c’.
`[abc]{5,}`	5 or more of ‘a’, ‘b’ or ‘c’.
`[abc]{,7}`	7 or fewer of ‘a’, ‘b’ or ‘c’.

We saw the plus modifier, “+”, meaning “one or more”. There are a couple of related modifiers that are often useful: a query, “?”, means zero or one of the pattern and asterisk, “*”, means “zero or more”. The more precise counting is done with curly brackets.

Note that in shell expansion of file names (“globbing”) the asterisk means “any string”. In regular expressions it means nothing on its own and is purely a modifier.

“Escaping” characters

Now let's pick up a few stray questions that might have arisen as we built our log file pattern. If square brackets identify sets of letters to match, what matches a square bracket? How would I match the literal string “`[abcde]`”, for example?

The way to mean “a real square bracket” is to precede it with a backslash. Generally speaking, if a character has a special meaning then preceding it with a backslash turns off that specialness. So “`[`” is special, but “`\`

[” means “just an open square bracket”. (Similarly, if we want to match a backslash we use “\\”.)

`[abcd]`	“Any one of ‘a’, ‘b’, ‘c’, or ‘d’.”
`\[abcd\]`	“[abcd]”

Generally speaking, if a character has a special meaning then preceding it with a backslash turns off that specialness. So “[” is special and “\[” means “just an open square bracket” (i.e. it has no special meaning).

Conversely, if a character is just a plain character then preceding it with a backslash can make it special. For example, “d” matches just the lower case letter “d” (i.e. it’s not special) but “\d” matches any one digit (and is special).

There is one last very important

If a character has no special meaning with a backslash in front of it then the backslash has no effect (but does not magically become a backslash itself. So “\␣” and “␣” both represent a single space.

Because regular expressions use backslashes and so does Python itself special care needs to be taken to stop the two clashing. Python has a special syntax to tell it not to treat backslashes specially. This allows the regular expression module direct access to them. Python’s special syntax is called “raw strings” and works by preceding a literal string with a “r” (for “raw”). Python treats backslashes in raw strings as just an ordinary character:

```
>>> print('\\')
\
>>> print(r'\\')
\\
>>>
```

We will start using raw strings in all our future examples and exercises:

```
pattern = r'XXXX'
```

Use raw strings for all regular expression patterns, even if you don’t strictly need to for a specific example. It’s just a good habit to get into.

Exercise 5

The scripts `exercise05a.py` and `exercise05b.py` differ only in that 5b uses a raw string to define the pattern:

```
pattern = r'\\ in'
```

Both read from the same file, `example05.txt`. Run them both and determine why the outputs differ. Make sure you can understand why `example5b.py` extracts the output it does.

We can add to the backslash letters now:

`\A`	`^`	Anchor start of line
`\Z`	`$`	Anchor end of line
`\d`	`[0-9]`	Any digit
`\D`	`[^0-9]`	Any non-digit
`\s`		Any white space character (space, tab, …)
`\S`		Any non-white-space character
`\w`	`[0-9a-zA-Z_]`	Any alphabetic, numeric or underscore character. (A “word character”.)

`\W`	`[^0-9a-zA-Z_]`	Any non-word character.

Our `atoms.log` file has a double space in each line. Guessing how many spaces there are is prone to error. The expression "`\s+`" means "some white space" and is very useful.

A real world example

You now know enough to write a significant regular expression. To demonstrate that you know enough to write a real world regular expression this extended exercise uses a log file from the real world (actually from the author's workstation) and is an example of just how you might use a regular expression in practice.

The file messages is part of the system log file (/var/log/messages) from a workstation and contains a number of lines indicating that a hacker attempted to break in to the system. Our job will be to find just those lines.

The lines we are looking for look like this:

```
Jun 30 03:02:14 noether sshd[9510]: Invalid user rpc from 65.19.189.149
Jun 30 03:02:16 noether sshd[9515]: Invalid user gopher from 65.19.189.149
Jul  1 07:41:11 noether sshd[14506]: Invalid user test from 210.51.172.168
```

and the lines we can ignore look like this:

```
Jun 26 23:20:33 noether syslog-ng[2298]: STATS: dropped 0
Jun 26 23:37:31 noether sshd[19439]: Accepted publickey for dump from
131.111.4.6 port 54086 ssh2
Jun 26 23:37:32 noether logger: starting incr /home
```

There is some obvious text to look for, "`Invalid user`", but we will write a regular expression that matches the entire line because this will lead on to our being able to extract just parts of the line. (We have to match them to extract them.) So this exercise is split into three parts.

Exercise 6

The script `exercise06.py` is a bare-bones filter script. Edit the pattern to just look for "`Invalid user`" with no anchoring or anything clever:

```
$ python3 exercise06.py

Jun 25 23:47:33 noether sshd[9277]: Invalid user account
from 207.54.140.124

…

Jul  1 07:41:34 noether sshd[14546]: Invalid user test from 210.51.172.168
```

You should get 1,366 lines of output:

```
$ python3 exercise06.py | wc -l

1366
```

Exercise 7

Copy `exercise06.py` to `exercise07.py` and edit the pattern to also match everything beyond the “Invalid user” text. Recall that the lines look like this:

```
Jun 30 03:02:16 noether sshd[9515]: Invalid user gopher
from 65.19.189.149
```

You need to extend the pattern to include:

- a positive number of non-white space characters for the varying user name (recall the counting modifiers and the backslash specials),
- the fixed text “`from`”, and
- the IP addresses (four lots of digits separated by full stops, and
- the end of line. (“`$`”)

We strongly recommend that you get the first extension (user name) right and then move on to the next and so on.
If successful you should get exactly the same number of lines through:

```
$ python3 exercise07.py | wc -l
1366
```

Exercise 8

Copy `exercise07.py` to `exercise08.py` and edit the pattern to everything up to the “Invalid user” text too.
Hint: work backwards from what you have already got working. Again, we strongly suggest that you do it one bit at a time.
You need to extend the pattern to include (working backwards):

- the closing square bracket, colon and space (fixed text),
- a sequence of digits
- the fixed text “`noether sshd[`”, (`noether` is the name of the workstation and the `sshd` is the service the hackers are trying to break in through),
- the time (pairs of digits separated by colons),
- the day of the month, (Note that the second character is a digit. The first is a digit or a space.)
- the month. (Treat this as a capital letter followed by two lower case letters.)
- the start of the line. (“`^`”)

If successful you should get exactly the same number of lines through:

```
$ python3 exercise08.py | wc -l
1366
```

Alternation

In the previous exercise we used an expression for the three letter month abbreviations of something like “[A-Z][a-z][a-z]” or ”[A-Z][a-z]{2}”. This is a little unsatisfactory in general. As well as matching “Jan” and “Feb” it matches “Fan” and “Jeb”. What we want to say is “one of the twelve valid three letter month abbreviations”.

Regular expressions have a syntax that says “this or this or this”. Such a choice is called an “alternation” and the syntax for it is this:

```
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)
```

The brackets hold the whole thing together as a single entity, called a “group”. The vertical bars, pronounced “or” separate the alternatives.

Groups

Groups do not have to contain alternations. They can be useful for other reasons to group an expression together.

The bracketed groups can be followed by any of the counting expressions. So "`([ABC][abc]){2}`" matches "AbCa" and "BcAa" but not "ABca" or "aAcC". For that example, we don't need to use groups; we could have used "`[ABC][abc][ABC][abc]`".

However, we can use them with counting to expression patterns that cannot be expressed in other ways. For example, "`([ABC][abc])+`" means "one or more repetitions of an upper case "A", "B", or "C" followed by a lower case "a", "b", or "c". It will match

Ab
AcBa
AaBbCc
AbCaBcAb
etc.

We will meet groups again, when e come to look at extracting components of matching strings.

Verbose regular expression language

Recall the regular expression you wrote for the last exercise:

```
^[A-Z][a-z]{2}␣[123␣][0-9]␣\d\d:\d\d:\d\d␣noether␣sshd\[\d+\]:␣
Invalid␣user␣\S+␣from␣\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$
```

This is about as complex a regular expression as you would want to encounter without some assistance from the language. If you struggled with the exercise, re-read this answer to see if you understand it.

If regular expressions were a programming language in their own right, we would expect to be able to lay them out sensibly to make them easier to read and to include comments.

Python allows us to do both of these with a special option to the `re.compile()` function which we will meet now.

(We might also expect to have variables with names, and we will come to that in this course too.)

Our fundamental problem is that the enormous regular expression we have just written runs the risk of becoming gibberish. It was a struggle to write and if you passed it to someone else it would be even more of a struggle to read. It gets even worse if you are asked to maintain it after not looking at it for six months.

We need to be able to spread it out over several lines so that how it breaks down into its component parts becomes clearer. It would be nice if we had comments so we could annotate it too.

Python's regular expression system has all this as an option and calls it, rather unfairly, "verbose mode".

The regular expression language we have met so far is fairly portable.
Verbose regular expressions are a Python-only version.

Let's start by splitting that regular expression over several lines for ease of reading. (This would be a Python raw long string.)

```
pattern = r'''
^
[A-Z][a-z]{2}␣
[123␣][0-9]␣
\d\d:\d\d:\d\d␣
noether␣sshd
\[\d+\]:␣
Invalid␣user␣
\S+␣
from␣
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

```
$
'''
```

This is no longer a useful regular expression. We will fix it up one stage at a time.

Ideally we would like to add comments to this multi-line layout and still have the regular expression work:

```
pattern = r'''
^
[A-Z][a-z]{2}␣                          # Month
[123␣][0-9]␣                            # Day
\d\d:\d\d:\d\d␣                         # Time
noether␣sshd
\[\d+\]:␣                               # Process ID
Invalid␣user␣
\S+␣                                    # Fake user ID
from␣
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}      # Host address
$
'''
```

We will use the hash character (“#”) for comments inside our verbose regular expressions just as we have in Python itself. Of course, if a hash introduces a comment in verbose regular expressions, what matches a hash character? In verbose regular expressions we use an escaped hash (i.e. preceded by a backslash) to match a literal hash character.

The next issue we hit concerns spaces. We want to match on spaces, and our original regular expression had spaces in it. However, any spaces between the end of the line and the start of any putative comments mustn't contribute towards the matching component.

We will need to treat spaces differently in the verbose version. Bring on the backslashes!

We will use “\␣” to represent a literal space (i.e. one we want to match) and will ignore any other spaces in the pattern (e.g. those between the regular expression and the comment).

Note that we don’t need to backslash a space or a hash in the square brackets.

Space ignored generally	`␣`
Backslash space recognised	`\␣`
Backslash not needed in [...]	`[123␣]`
Hash introduces a comment	`# Month`
Backslash hash matches “#”	`\#`
Backslash not needed in [...]	`[123#]`

This gives us a pattern like this:

```
pattern = r'''
^
[A-Z][a-z]{2}\␣                         # Month
[123␣][0-9]\␣                           # Day
\d\d:\d\d:\d\d␣                         # Time
noether\␣sshd
\[\d+\]:\␣                              # Process ID
Invalid\␣user\␣
\S+\␣                                   # Fake user ID
from\␣
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}      # Host address
$
'''
```

So now all we have to do is to tell Python to use this verbose mode instead of its usual one. We do this as an option on the `re.compile()` function just as we did when we told it to work case insensitively. There is a Python module constant `re.VERBOSE` which we use in exactly the same way as we did `re.IGNORECASE`. It has a short name “`re.X`” too.

Incidentally, if you ever wanted case insensitivity and verbosity, you add the two together:
`regexp = re.compile(pattern, re.IGNORECASE+re.VERBOSE)`

So how would we change a filter script in practice to use verbose regular expressions? It's actually a very easy four step process.

1. Convert your pattern string into a multi-line string.
2. Make the backslash tweaks necessary.
3. Change the `re.compile()` call to have the `re.VERBOSE` option.
4. Test your script to see if it still works!

Exercise 9

Copy `exercise08.py` to `exercise09.py` and convert its regular expression into the verbose pattern derived above.
If successful you should get exactly the same output as from `exercise08.py`.

```
$ python3 exercise08.py
```

Exercise 10

Copy `exercise03.py` (the `atoms.log` filter) to `exercise10.py` and convert its regular expression into a verbose format (with comments).
If successful you should get exactly the same output as from `exercise03.py`.

```
$ python3 exercise10.py
```

Components of matching text

We're almost finished with the regular expression syntax now. We have most of what we need for this course and can now get on with developing Python's system for using it. We will continue to use the verbose version of the regular expressions as it is easier to read, which is helpful for courses as well as for real life!

Let's look at the verbose pattern for our `atoms.log` file again:

```
pattern = r'''
^                                   # Start of line
RUN\␣
\d{6}                               # Job number
\␣COMPLETED\.\␣\␣OUTPUT\␣IN\␣FILE\␣
[a-z]+\.dat                         # File name

\.
$                                   # End of line
'''
```

Suppose we have a line that matches, but we want to know the job number and file name that were in the matching line. How do we do this?

What we will do is to label the two components in the pattern and then look at Python's mechanism to get at their values.

We start by changing the pattern to place parentheses (round brackets) around the two components of interest to make them groups.

```
pattern = r'''
^                                   # Start of line
RUN\␣
(\d{6})                             # Job number
\␣COMPLETED\.\␣\␣OUTPUT\␣IN\␣FILE\␣
```

```
([a-z]+\.dat)                          # File name
\.
$                                      # End of line
'''
```

So

```
\d{6}          →    (\d{6})
[a-z]+\.dat    →    ([a-z]+\.dat)
```

Now we are asking for certain parts of the pattern to be specially treated (as “groups”) we must turn our attention to the result of the search to get at those groups.

To date all we have done with the results is to test them for truth or falsehood: “does it match or not?” Now we will dig more deeply.

```
regexp = re.compile(pattern,re.VERBOSE)
for line in sys.stdin:
    result = regexp.search(line)
    if result:
        ...
```

The match object, “`result`”, has a method that gives access to the groups within its defining pattern, `result.group()`.

So if the line that matches is this line:

RUN 000001 COMPLETED. OUTPUT IN FILE hydrogen.dat.

then

```
result.group(1)  →    '000001'
result.group(2)  →    'hydrogen.dat'
result.group(0)  →    whole line
```

Note that `group(0)` gives the entire text that the pattern matched. It’s only because we have a pattern anchored at the start and end of the line that this is the entire line.

So now we can write out just those elements of the matching lines that we are interested in:

```
if result:
    print(result.group(1), result.group(2))
```

(We have switched to `print()` from `sys.stdout.write()` just to keep the lines a bit shorter.)

Note that we still have to test the result variable to make sure that it is not `None` (i.e. that the regular expression matched the line at all). This is what the `if`... test does because `None` tests `False`. We cannot ask for the `group()` method on `None` because it doesn't have one. If you make this mistake you will get an error message “`AttributeError: 'NoneType' object has no attribute 'group'`” and your script will terminate abruptly.

Exercise 11

Copy your script `exercise09.py` to `exercise11.py` and edit it to add brackets to set groups for the account name and the remote address. So, for a line like this:

```
Jun 25 23:47:56 noether sshd[9367]: Invalid user alex from
207.54.140.124
```

you would capture “`alex`” and “`207.54.140.124`”.
Then modify the output to print just those two values.

Named groups

Groups in regular expressions are good but they're not perfect. They suffer from the sort of problem that creeps up on you only after you've been doing Python regular expressions for a bit.

Suppose you decide you need to capture another group within a regular expression. If it is inserted between the first and second existing group, say, then the old group number 2 becomes the new number 3, the old 3 the new 4 and so on.

There's also a problem that "`results.group(2)`" doesn't shout out what the second group actually was.

There's a solution to this: We will associate *names* with groups rather than just numbers.

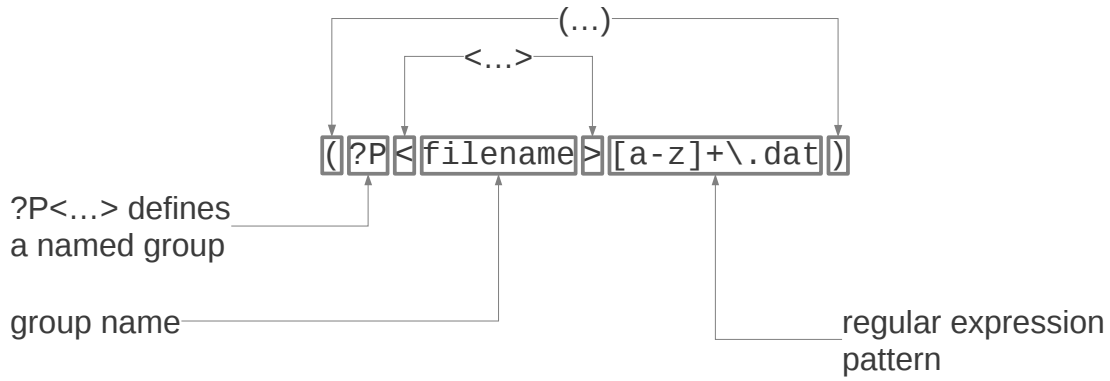
Consider the regular expression we have been using for our `atoms.log` searching. We currently have two groups for the job number and the file name.

So how do we do this naming?

We insert some additional controls immediately after the open parenthesis. In general in Python's regular expression syntax "`(?`" introduces something special that may not even be a group (though in this case it is). We specify the name with the rather bizarre syntax "`?P<`*`groupname`*`>`":

```
(\d{6})          →    (?P<jobnum>\d{6})
([a-z]+\.dat)    →    (?P<filename>[a-z]+\.dat)
```

So we name the job number (six digits) "`jobnum`" and the file name "`filename`".



So the group is defined as usual by parentheses (round brackets). Next must come "?P" to indicate that we are handling a named group. Then comes the name of the group in angle brackets. Finally comes the pattern that actually does the matching. None of the "?P<...>" business is used for matching; it is purely for naming.

To refer to a group by its name, you simply pass the name to the `group()` method as a string. You can still also refer to the group by its number. So in the example here, `result.group('jobno')` is the same as `result.group(1)`, since the first group is named "`jobno`".:

RUN 000001 COMPLETED. OUTPUT IN FILE hydrogen.dat.

gives

```
result.group('jobno')        →        '000001'
result.group('filename')     →        'hydrogen.dat'
```

So we can change the print statement to use these names:

```
if result:
    print(result.group('jobno'), result.group('filename'))
```

Exercise 12

Copy your script `exercise11.py` to `exercise12.py` and edit name the groups "login" and "address". Print them using these names.

```
Jun 25 23:47:56 noether sshd[9367]: Invalid user alex from
207.54.140.124
```

```
login → alex
address → 207.54.140.124
```

Ambiguity and greed

Input: `/usr/share/dict/words`

Reg. exp.: `^([a-z]+)([a-z]+)$`

Script: `ambiguous.py`

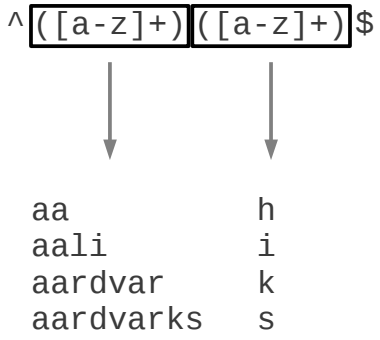
Question: What part of the word goes in group 1, and what part goes in group 2?

Groups are all well and good, but are they necessarily well-defined? What happens if a line can fit into groups in two different ways?

For example, consider the list of words in `/usr/share/dict/words`. The lower case words in this line all match the regular expression "[a-z]+[a-z]+" because it is a series of lower case letters followed by a series of lower case letters. But if we assign groups to these parts, "`([a-z]+)([a-z]+)`", which part of the word goes into the first group and which in the second?

You can find out by running the script `ambiguous.py` which is currently in your home directory:

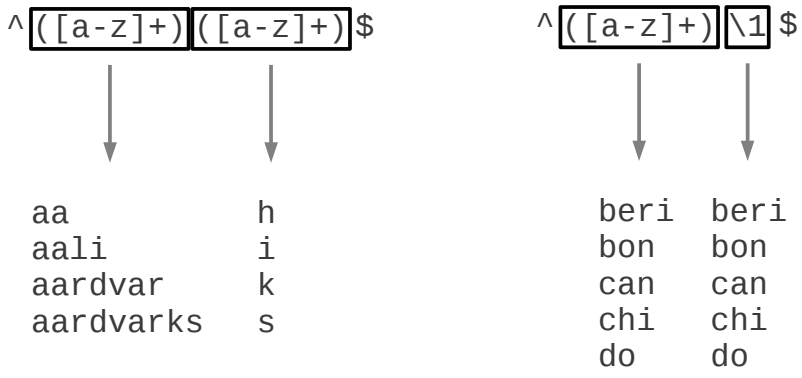
```
$ python3 ambiguous.py
aa h
aahe d
aahin g
aah s
aa l
aali i
aalii s
aal s
aardvar k
...
```



Python's implementation of regular expressions makes the first group "greedy"; the first group swallows as many letters as it can at the expense of the second. There is no guarantee that other languages'

implementations will do the same, though. You should always aim to avoid this sort of ambiguity. You can change the greed of various groups with yet more use of the query character but please note the ambiguity caution above. If you find yourself wanting to play with the greediness you're almost certainly doing something wrong at a deeper level.

Internal references



In our ambiguity example, `ambiguous.py`, we had the same pattern, "`[a-z]+`", repeated twice. These then matched against different strings. The first matched against "`aardvar`" and the second against "`k`", for example. How can we say that we want the same string twice?

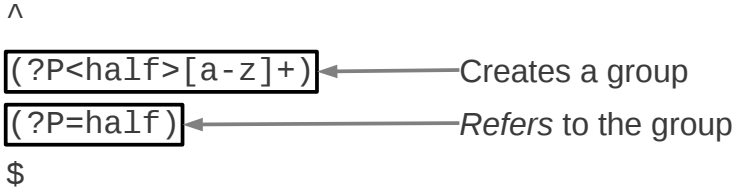
Now that we have groups in our regular expression we can use them for this purpose. So far the bracketing to create groups has been purely labelling, to select sections we can extract later. Now we will use them within the expression itself.

We can use a backslash in front of a number (for integers from 1 to 99) to mean "that number group in the current expression". The pattern "`^([a-z]+)\1$`" matches any string which is followed by the string itself again.

The file `double1.py` has an example of this working:

```
$ python3 double1.py

beri beri
bon bon
can can
chi chi
do do
hots hots
ma ma
mur mur
muu muu
pa pa
paw paw
pom pom
tar tar
tes tes
tu tu
```



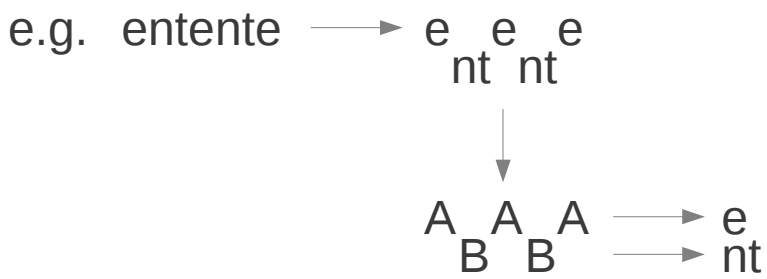
If we have given names to our groups, then we use the special Python syntax "`(?P=groupname)`" to mean "the group `groupname` in the current expression". So "`^(?P<half>[a-z]+)(?P=half)$`" matches any string which is the same sequence of lower case letters repeated twice.

Note that in this case the `(?...)` expression does not create a group; instead, it refers to one that already exists. Observe that there is no pattern language in that second pair of parentheses.

The script `double2.py` uses named groups.

Exercise 13

Copy the script `double2.py` to `exercise13.py` and adapt it to look for words with the pattern ABABA:



You should get "alfalfa" and "entente".

Crib sheet

`\A`	`^`	Anchor start of line
`\Z`	`$`	Anchor end of line
`\d`	`[0-9]`	Any digit
`\D`	`[^0-9]`	Any non-digit
`\s`		Any white space character (space, tab, …)
`\S`		Any non-white-space character
`\w`	`[0-9a-zA-Z_]`	Any alphabetic, numeric or underscore character. (A “word character”.)
`\W`	`[^0-9a-zA-Z_]`	Any non-word character.
`[abc]`		Any one of ‘a’, ‘b’ or ‘c’.
`[abc]+`		One or more ‘a’, ‘b’ or ‘c’.
`[abc]?`		Zero or one ‘a’, ‘b’ or ‘c’.
`[abc]*`		Zero or more ‘a’, ‘b’ or ‘c’.
`[abc]{6}`		Exactly 6 of ‘a’, ‘b’ or ‘c’.
`[abc]{5,7}`		5, 6 or 7 of ‘a’, ‘b’ or ‘c’.
`[abc]{5,}`		5 or more of ‘a’, ‘b’ or ‘c’.
`[abc]{,7}`		7 or fewer of ‘a’, ‘b’ or ‘c’.
`(…)`		A numbered group (no name)
`(…|…|…)`		Alternation
`(?P<name>…)`		A named group
`\1`, `\2`, …		Reference to numbered group
`(?P=name)`		Reference to named group