Python: Regular Expressions

Bruce Beckles

Bob Dowling

University Computing Service

Scientific Computing Support e-mail address:
scientific-computing@ucs.cam.ac.uk

UCS

Welcome to the University Computing Service's "Python: Regular Expressions" course.

The official UCS e-mail address for all scientific computing support queries, including any questions about this course, is:

scientific-computing@ucs.cam.ac.uk

Before we start, let's specify just what is and isn't in this course.

This course is a very simple, beginner's course on regular expressions. It mostly covers how to get Python to use them.

There is an on-line introduction called the Python "Regular Expression HowTo" at:

`http://docs.python.org/howto/regex`

and the formal Python documentation at

`http://docs.python.org/library/re.html`

There is a good book on regular expressions in the O'Reilly series called "Mastering Regular Expressions" by Jeffrey E. F. Freidl. Be sure to get the third edition (or later) as its author has added a lot of useful information since the second edition. There are details of this book at:

`http://regex.info/`

There is also a Wikipedia page on regular expressions which has useful information itself buried within it and a further set of references at the end:

`http://en.wikipedia.org/wiki/Regular_Expression`

A regular expression is a “pattern” describing some text:

“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+`

UCS

3

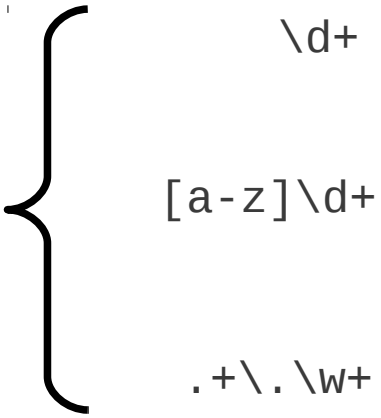
A regular expression is simply some means to write down a pattern describing some text. (There is a formal mathematical definition but we’re not bothering with that here. What the computing world calls regular expressions and what the strict mathematical grammarians call regular expressions are slightly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

For example we might like to say “a series of digits” or a “a single lower case letter followed by some digits”. There are terms in regular expression language for all of these concepts.

A regular expression is a “pattern” describing some text:

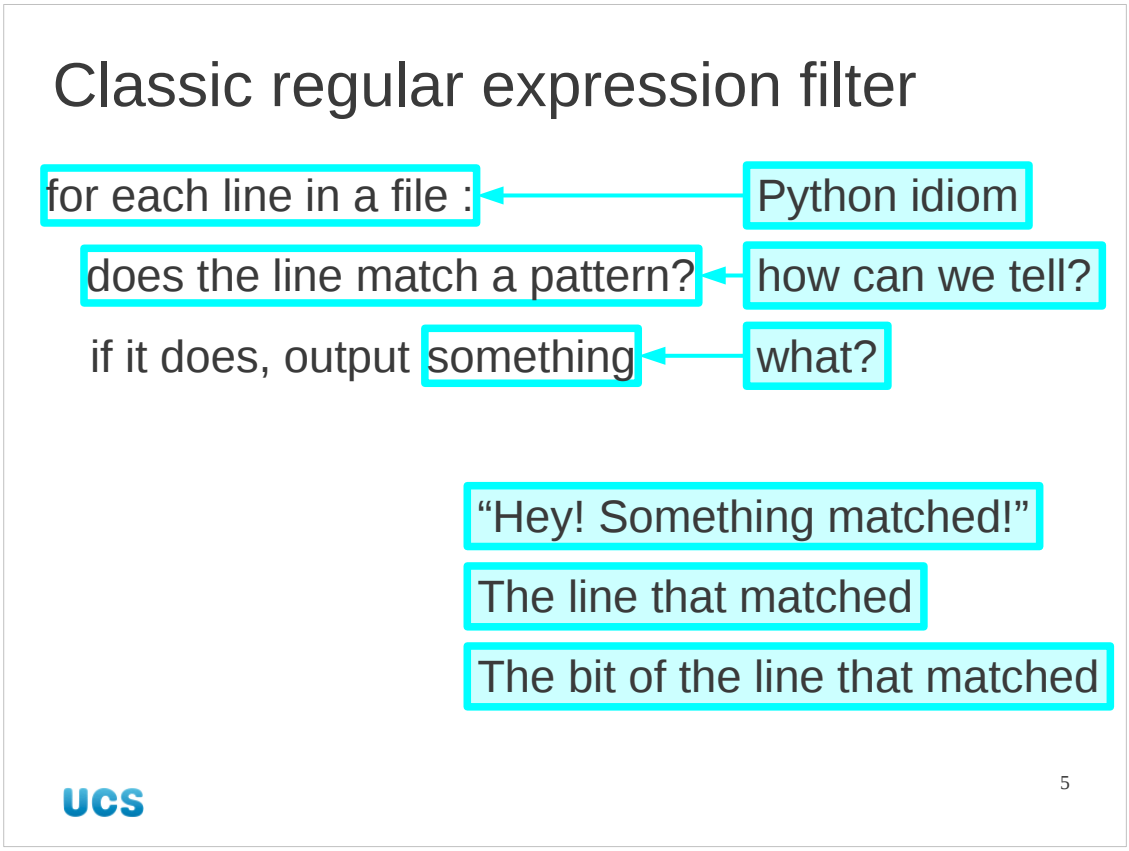
Isn't this just gibberish?

The language of regular expressions



UCS

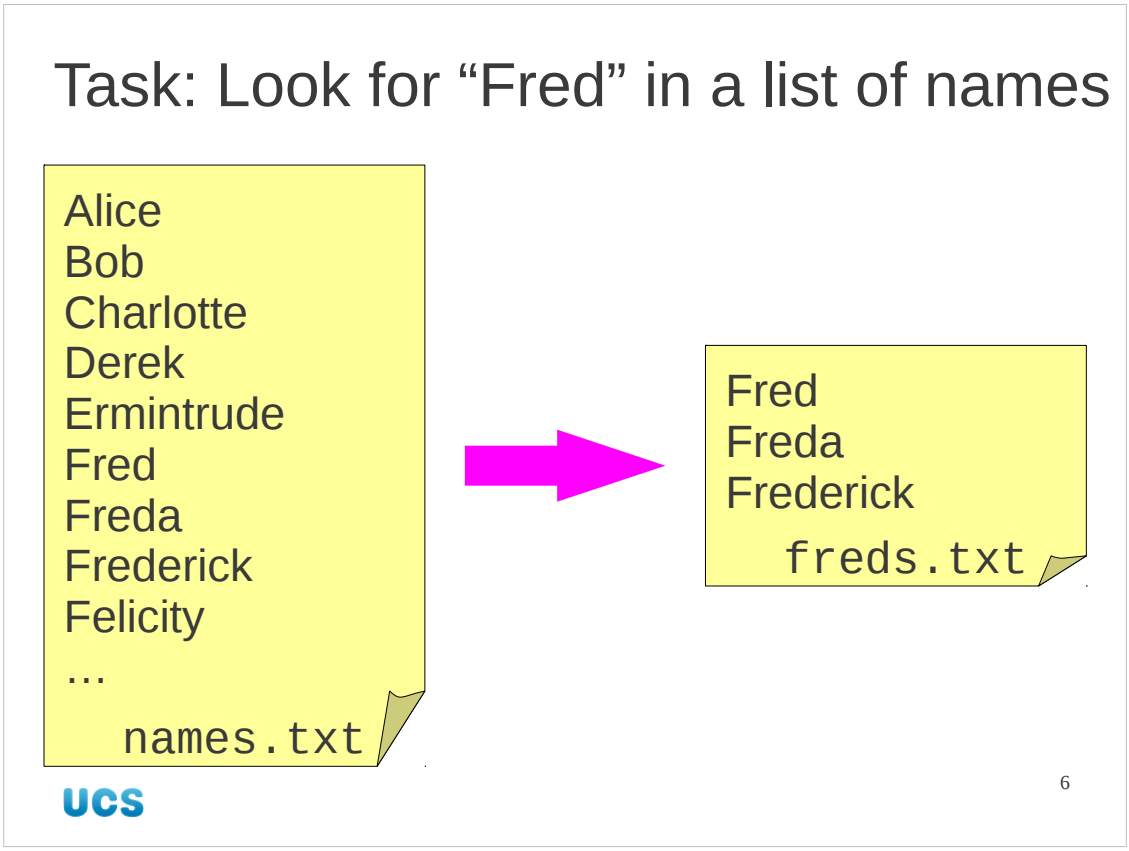
We will cover what this means in a few slides time. We will start with a “trivial” regular expression, however, which simply matches a fixed bit of text.



This is a course on using regular expressions from Python, so before we introduce even our most trivial expression we should look at how Python drives the regular expression system.

Our basic script for this course will run through a file, a line at a time, and compare the line against some regular expression. If the line matches the regular expression the script will output something. That “something” might be just a notice that it happened (or a line number, or a count of lines matched, etc.) or it might be the line itself. Finally, it might be just the bit of the line that matched.

Programs like this, that produce a line of output if a line of input matches some condition and no line of output if it doesn't are called "filters".



So we will start with a script that looks for the fixed text “`Fred`” in the file `names.txt`. For each line that matches, the line is printed. For each line that doesn't nothing is printed.

c.f. grep

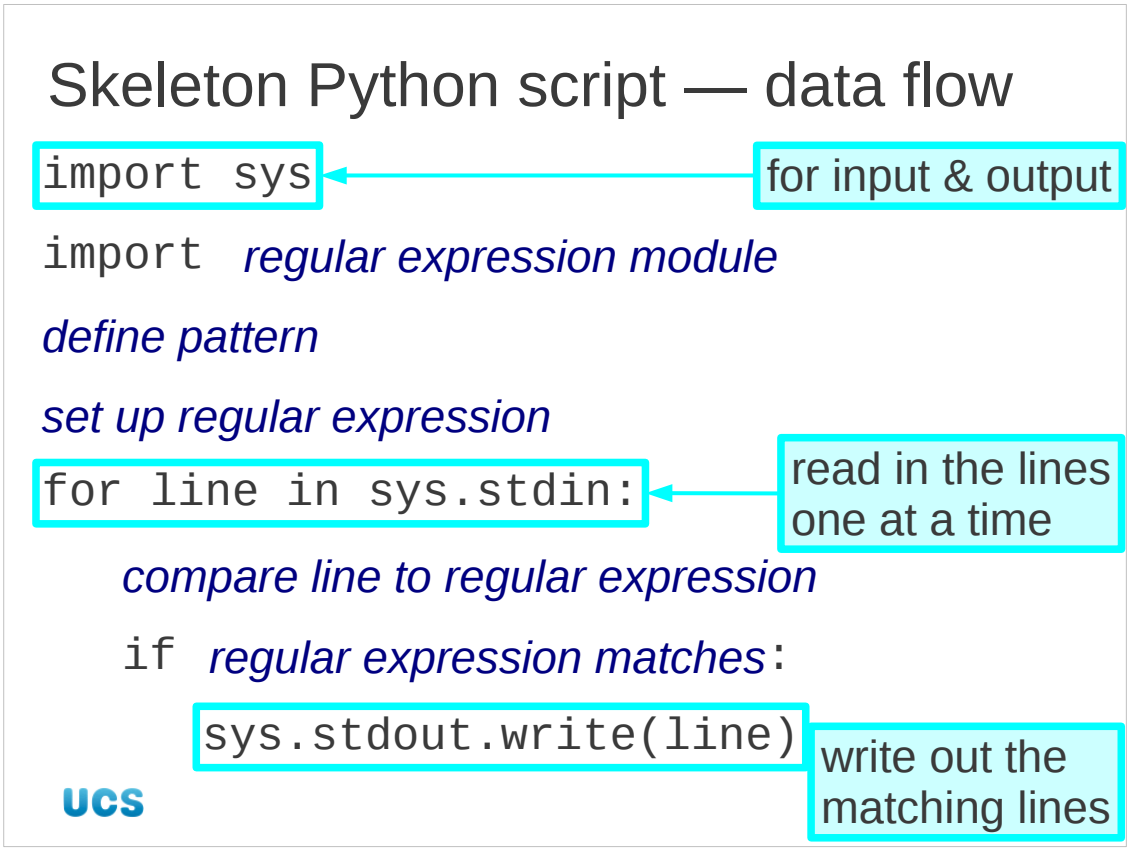
```
$ grep 'Fred' < names.txt
Fred
Freda
Frederick
$
```

(Don't panic if you're not a Unix user.)

UCS

7

This is equivalent to the traditional Unix command, `grep`.
Don't panic if you're not a Unix user. This is a Python course, not a Unix one.



So we will start with the outline of a Python script and review the non-regular expression lines first.

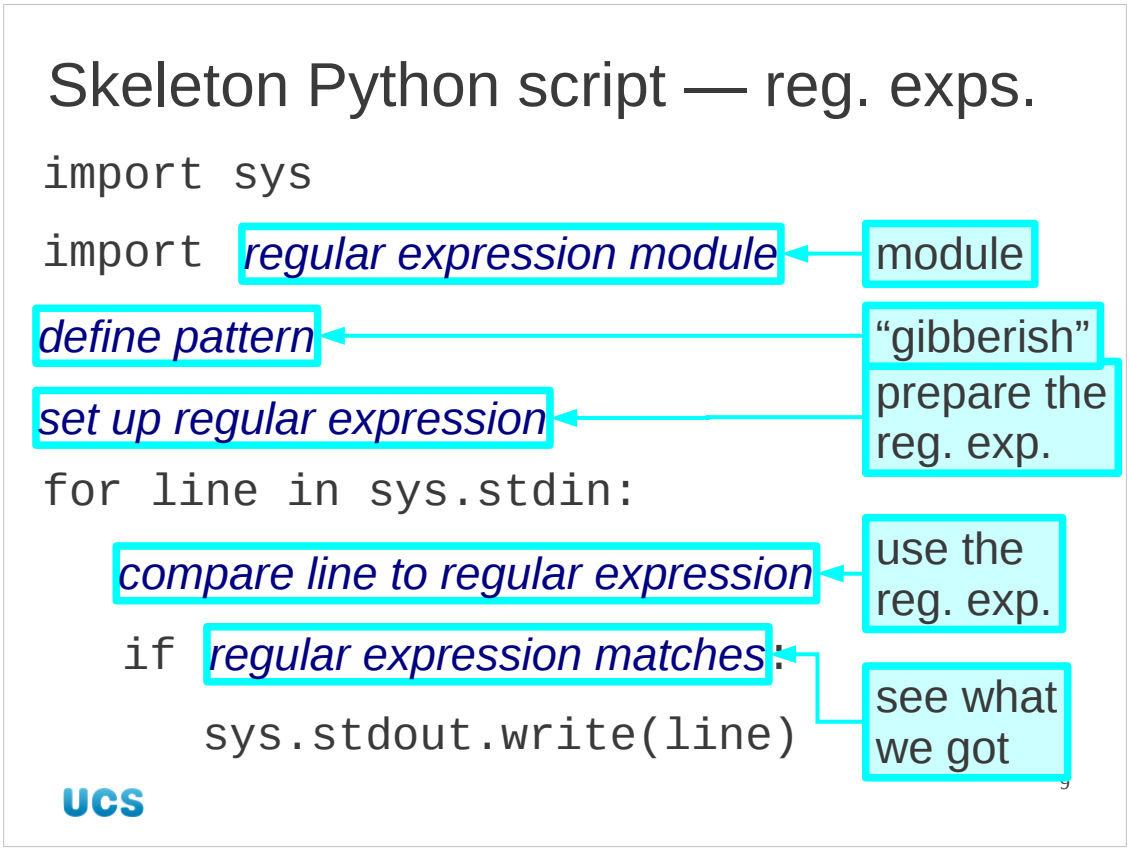
Because we are using standard input and standard output, we will import the `sys` module to give us `sys.stdin` and `sys.stdout`.

We will process the file a line at a time. The Python object `sys.stdin` corresponds to the standard input of the program and if we use it like a list, as we do here, then it behaves like the list of lines in the file. So the Python "`for line in sys.stdin:`" sets up a `for` loop running through a line at a time, setting the variable `line` to be one line of the file after another as the loop repeats. The loop ends when there are no more lines in the file to read.

The `if` statement simply looks at the results of the comparison to see if it was a successful comparison for this particular value of `line` or not.

The `sys.stdout.write()` line in the script simply prints the line. We could just use `print` but we will use `sys.stdout` for symmetry with `sys.stdin`.

The pseudo-script on the slide contains all the non-regular-expression code required. What we have to do now is to fill in the rest: the regular expression components.



Now let's look at the regular expression lines we need to complete.

Python's **r**egular **e**xpression handling is contained in a module so we will have to import that.

We will need to write the "gibberish" that describes the text we are looking for.

We need to set up the regular expression in advance of using it. (Actually that's not always true but this pattern is more flexible and more efficient so we'll focus on it in this course.)

Finally, for each line we read in we need some way to determine whether our regular expression matches that line or not.

Loading the module

The Python module for handling **r**egular **e**xpressions is called “`re`”.

Skeleton Python script — 1

```
import sys
import re
```

Ready to use regular expressions

define pattern

set up regular expression

```
for line in sys.stdin:
```

compare line to regular expression

```
    if regular expression matches:
        sys.stdout.write(line)
```

UCS

11

So we add that line to our script.

Defining the pattern

UCS

In this very simple case of looking for an exact string, the pattern is simply that string. So, given that we are looking for "Fred", we set the pattern to be "Fred".

Skeleton Python script — 2

```
import sys
import re
pattern = "Fred"
```

Define the pattern

set up regular expression

```
for line in sys.stdin:
```

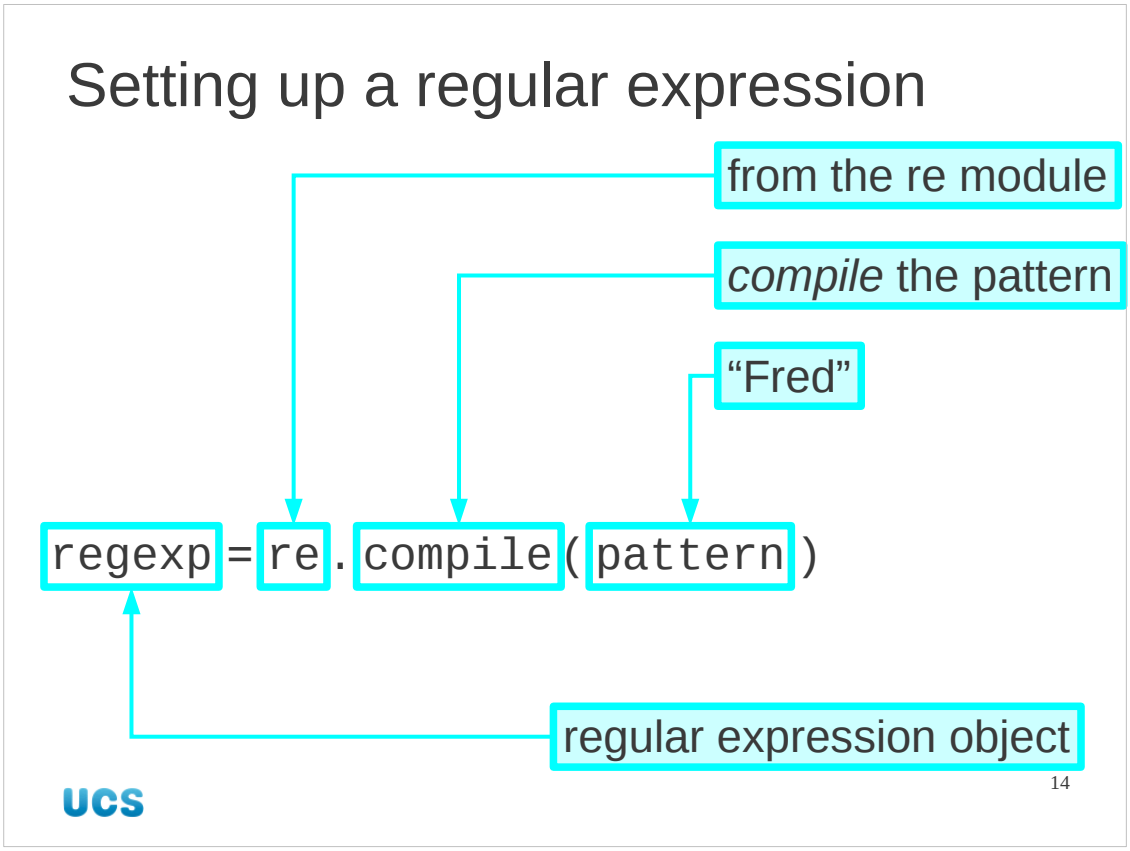
compare line to regular expression

```
    if regular expression matches:
        sys.stdout.write(line)
```

UCS

13

We add this line to our script, but this is just a Python string. We still need to turn it into something that can do the searching for "Fred".



Next we need to look at how to use a function from this module to set up a regular expression object in Python from that simple string.

The `re` module has a function “`compile()`” which takes this string and creates an object Python can do something with. This is deliberately the same word as we use for the processing of source code (text) into machine code (program). Here we are taking a pattern (text) and turning it into the mini-program that does the testing.

The result of this compilation is a “regular expression object”, the mini program that will do work relevant to the particular pattern “`Fred`”. We assign the name “`regexp`” to this object so we can use it later in the script.

Skeleton Python script — 3

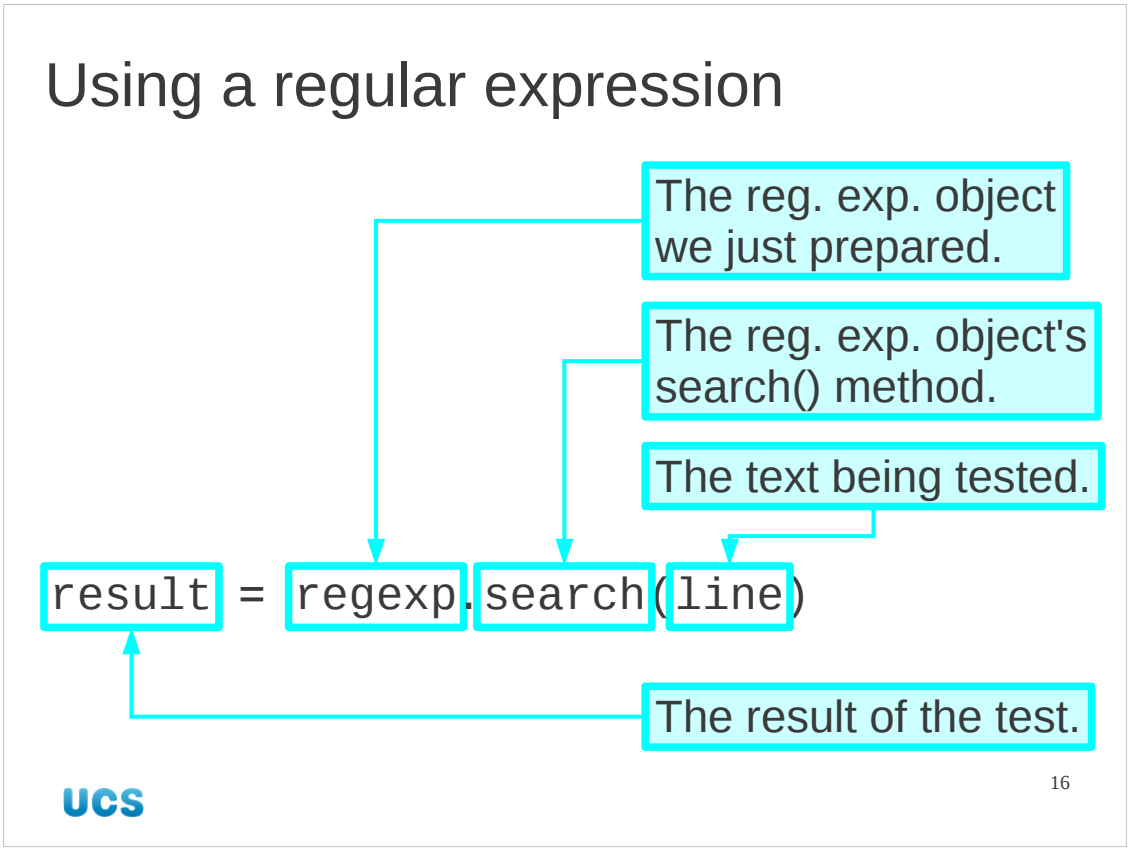
```
import sys
import re
pattern = "Fred"
regexp = re.compile(pattern)
for line in sys.stdin:
    compare line to regular expression
    if regular expression matches:
        sys.stdout.write(line)
```

Prepare the regular expression

UCS

15

So we put that compilation line in our script instead of our placeholder.
Next we have to apply that regular expression object, `regexp`, to each line as it is read in to see if the line matches.



We start by doing the test and then we will look at the test's results.

The regular expression object that we have just created, "`regexp`", has a method (a built in function specific to itself) called "`search()`". So to reference it in our script we need to refer to "`regexp.search()`". This method takes the text being tested (our input line in this case) as its only argument. The input line in in variable `line` so we need to run "`regexp.search(line)`" to get our result.

Note that the string "`Fred`" appears nowhere in this line. It is built in to the `regexp` object.

Incidentally, there is a related confusingly similar method called "`match()`". Don't use it. (And that's the only time it will be mentioned in this course.)

Skeleton Python script — 4

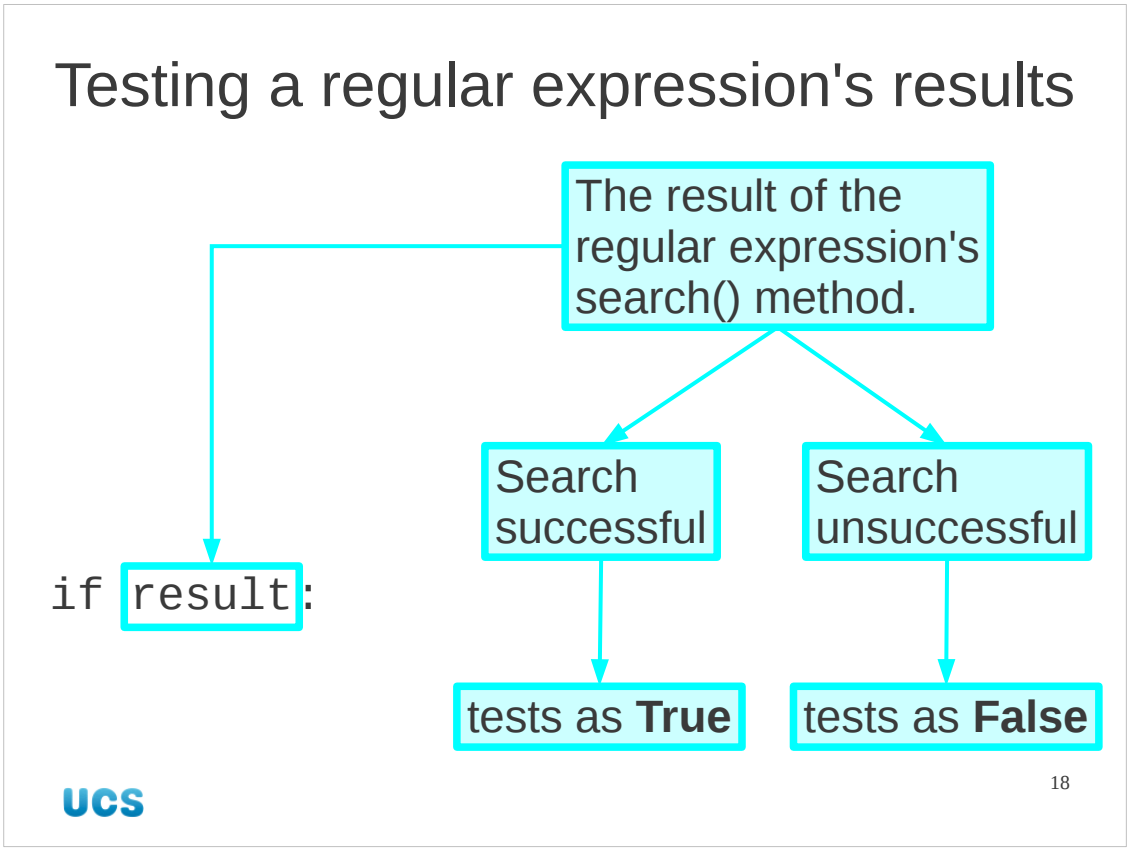
```
import sys
import re
pattern = "Fred"
regexp = re.compile(pattern)
for line in sys.stdin:
    result = regexp.search(line)
    if regular expression matches:
        sys.stdout.write(line)
```

Use the reg. exp.

UCS

17

So we put that search line in our script instead of our placeholder.
Next we have to test the result to see if the search was successful.



The `search()` method returns the Python “null object”, `None`, if there is no match and something else (which we will return to later) if there is one. So the `result` variable now refers to whatever it was that `search()` returned.

`None` is Python’s way of representing “nothing”. The `if` test in Python treats `None` as `False` and the “something else” as `True` so we can use `result` to provide us with a simple test.

Skeleton Python script — 5

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

See if the line matched

UCS

19

So if we drop that line into our skeleton Python script we have completed it.

This Python script is the fairly generic filter. If a input line matches the pattern write the line out. If it doesn't don't write anything.

We will only see two variants of this script in the entire course: in one we only print out certain parts of the line and in the other we allow for there being multiple Freds in a single line.

Exercise 1(a): complete the file

```
import sys
import re

pattern = "Fred"
regexp = …

for line in sys.stdin:
    result = …
    if … :
        sys.stdout.write(line)
```

filter01.py

5 mins

20

UCS

If you look in the directory prepared for you, you will find a Python script called "`filter01.py`" which contains just this script with a few critical elements missing. Your first exercise is to edit that file to make it a search for the string '`Fred`'.

Once you have completed the file, test it.

Exercise 1(b): test your file

```
$ python filter01.py < names.txt
Fred
Freda
Frederick
```

UCS

Note that three names match the test pattern: Fred, Freda and Frederick. If you don't get this result go back to the script and correct it.

Case sensitive matching

`names.txt`

Fred ✔
Freda ✔
Frederick ✔
Manfred ✘

Python matches are case sensitive by default

UCS

Note that it did not pick out the name “Manfred” also in the file. Python regular expressions are case sensitive by default; they do not equate “F” with “f”.

Case insensitive matching

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

Options are given as module constants:

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

case insensitive matching

and other options (some of which we'll meet later).

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

UCS

23

We can build ourselves a case *in*sensitive 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 `I` constant is a synonym for the `IGNORECASE` constant, so we can use "`re.I`" instead of "`re.IGNORECASE`" if we wish. We use `re.I` in the slide just to make the text fit, but generally we would 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.

Exercise 2: modify the script

1. Copy `filter01.py` → `filter02.py`
2. Edit `filter02.py`

   Make the search case insensitive.
3. Run `filter02.py`

```
$ python filter02.py < names.txt
```

5 mins

24

UCS

Copy your answer to the previous exercise into a new file called "`filter02.py`".

Edit this new file to make the search case insensitive. This involves a single modification to the `compile()` line.

Then run the new, edited script to see different results.

```
$ cp filter01.py filter02.py
$ gedit filter02.py
$ python filter02.py < names.txt
Fred
Freda
Frederick
Manfred
```

Serious example: Post-processing program output

RUN 000001 COMPLETED. OUTPUT IN FILE hydrogen.dat.
RUN 000002 COMPLETED. OUTPUT IN FILE helium.dat.
…
RUN 000039 COMPLETED. OUTPUT IN FILE yttrium.dat. 1 UNDERFLOW WARNING.
RUN 000040 COMPLETED. OUTPUT IN FILE zirconium.dat. 2 UNDERFLOW WARNINGS.
…
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.
…

UCS `atoms.log` 25

Now let's look at a more serious example.

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.

What do we want?

The file names for the runs with
no warning or error messages.

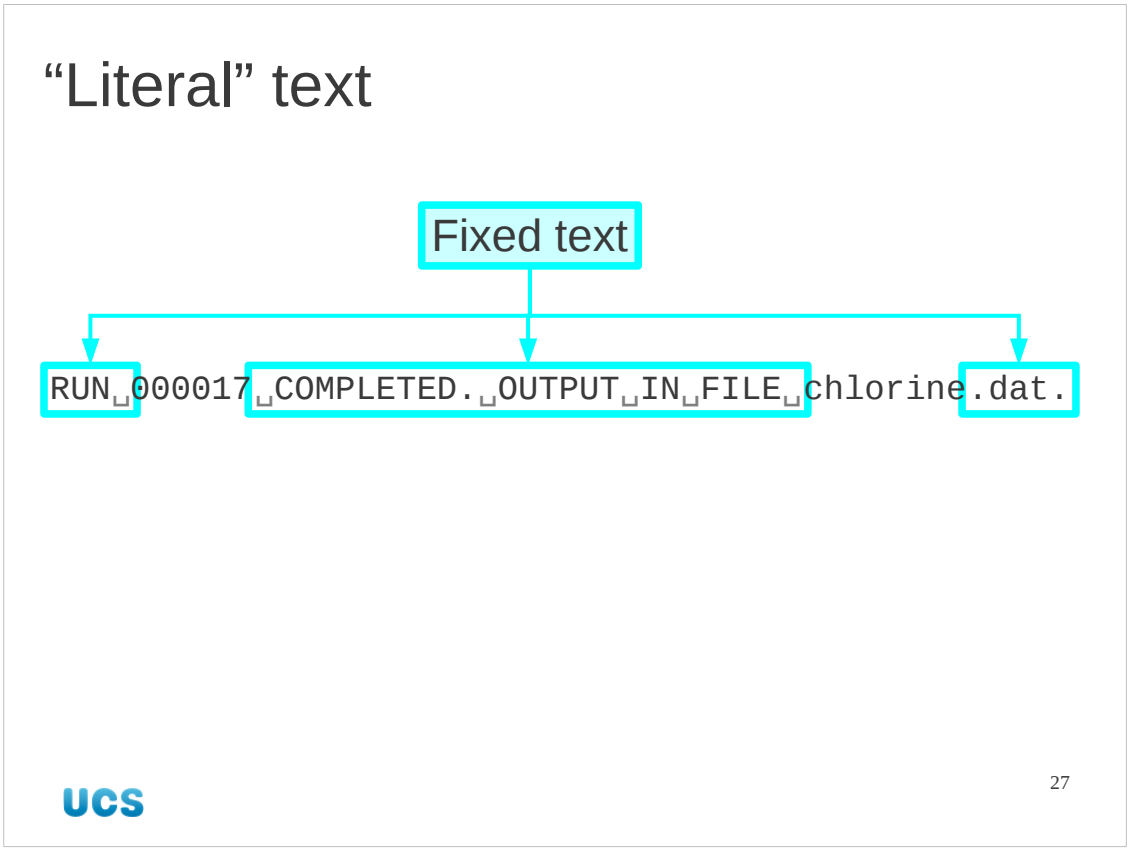
```
RUN␣000016␣COMPLETED.␣OUTPUT␣IN␣FILE␣sulphur.dat.
RUN␣000017␣COMPLETED.␣OUTPUT␣IN␣FILE␣chlorine.dat.
RUN␣000018␣COMPLETED.␣OUTPUT␣IN␣FILE␣argon.dat.
```

What *pattern* does this require?

UCS

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 are marking them explicitly in the slides.



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

Digits

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

Six digits

UCS

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”.

Letters

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

Sequence of lower case letters

UCS

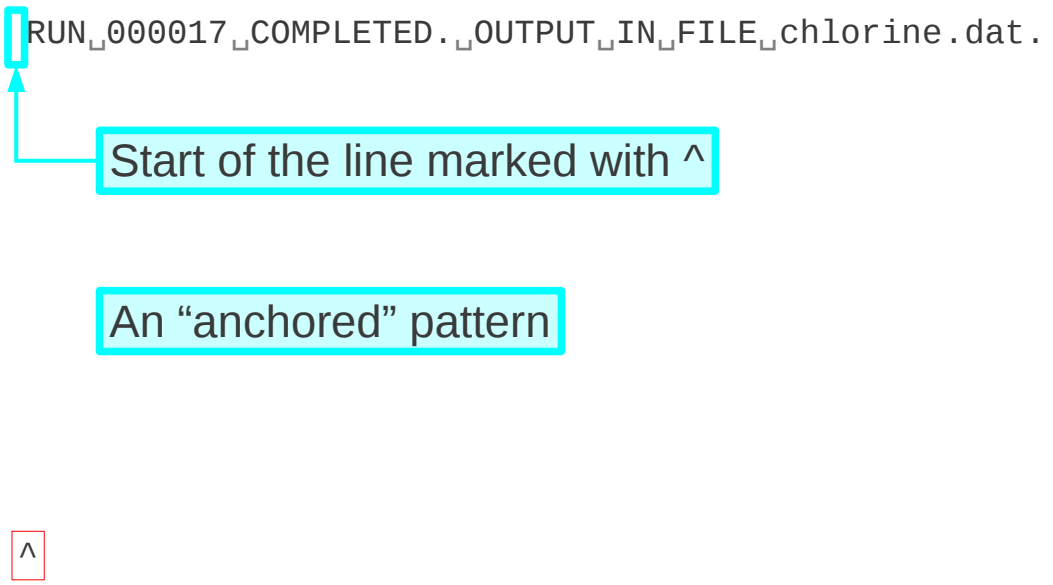
29

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.

Building the pattern — 1



^

UCS

31

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, “^”.

^ 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.

Building the pattern — 2

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

Literal text

Don't forget the space!

```
^RUN␣
```

UCS

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.

Building the pattern — 3

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

Six digits

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

`\d` “any digit”

```
^RUN \d\d\d\d\d\d
```

inelegant

UCS

33

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”.

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

Building the pattern — 4

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

Six digits

`\d`	“any digit”
`\d{6}`	“six digits”
`\d{5,7}`	“five, six or seven digits”

```
^RUN \d{6}
```

UCS

Regular expression pattern language has a solution to that 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)

Building the pattern — 5

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

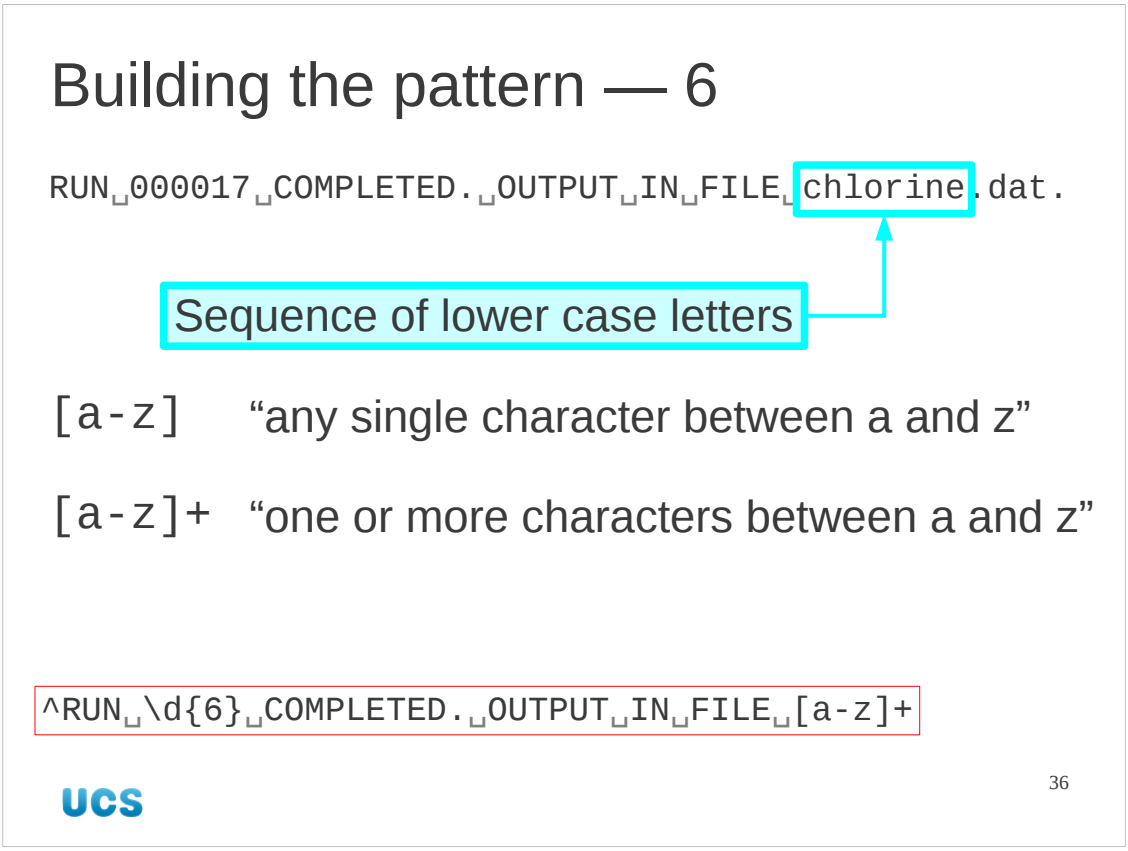
Literal text (with spaces).

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

UCS

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

Building the pattern — 6



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,}`”.

Building the pattern — 7

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

Literal text

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

UCS

37

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.)

Building the pattern — 8

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

End of line marked with $

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

UCS

38

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(a): running the filter

1. Copy `filter01.py` → `filter03.py`

2. Edit `filter03.py`

   Use the `^RUN…` regular expression.

3. Test it against `atoms.log`

```
$ python filter03.py < atoms.log
```

5 mins

39

UCS

You should try this regular expression for yourselves and get your fingers used to typing some of the strange sequences.

Copy the `filter01.py` file that you developed previously to a new file called `filter03.py`.

Then edit the simple “Fred” string to the new expression we have developed. This search should be case *sensitive*.

Then try it out for real.

```
$ cp filter01.py filter03.py
$ gedit filter03.py
$ python filter03.py < atoms.log
```

If it doesn't work, go back and fix `filter03.py` until it does.

Exercise 3(b): changing the filter

4. Edit `filter03.py`

   Lose the `$` at the end of the pattern.

5. What output do you think you will get this time?

6. Test it against `atoms.log` again.

   ```
   $ python filter03.py < atoms.log
   ```

7. Put the `$` back.


5 mins

UCS

40

Then change the regular expression very slightly simply by removing the final dollar character that anchors the expression to the end of the line.

What extra lines do you think it will match now.

Try the script again. Were you right?

Special codes in regular expressions

\A	**^**	Anchor start of line
\Z	**$**	Anchor end of line
\d		Any **d**igit
\D		Any non-digit

UCS

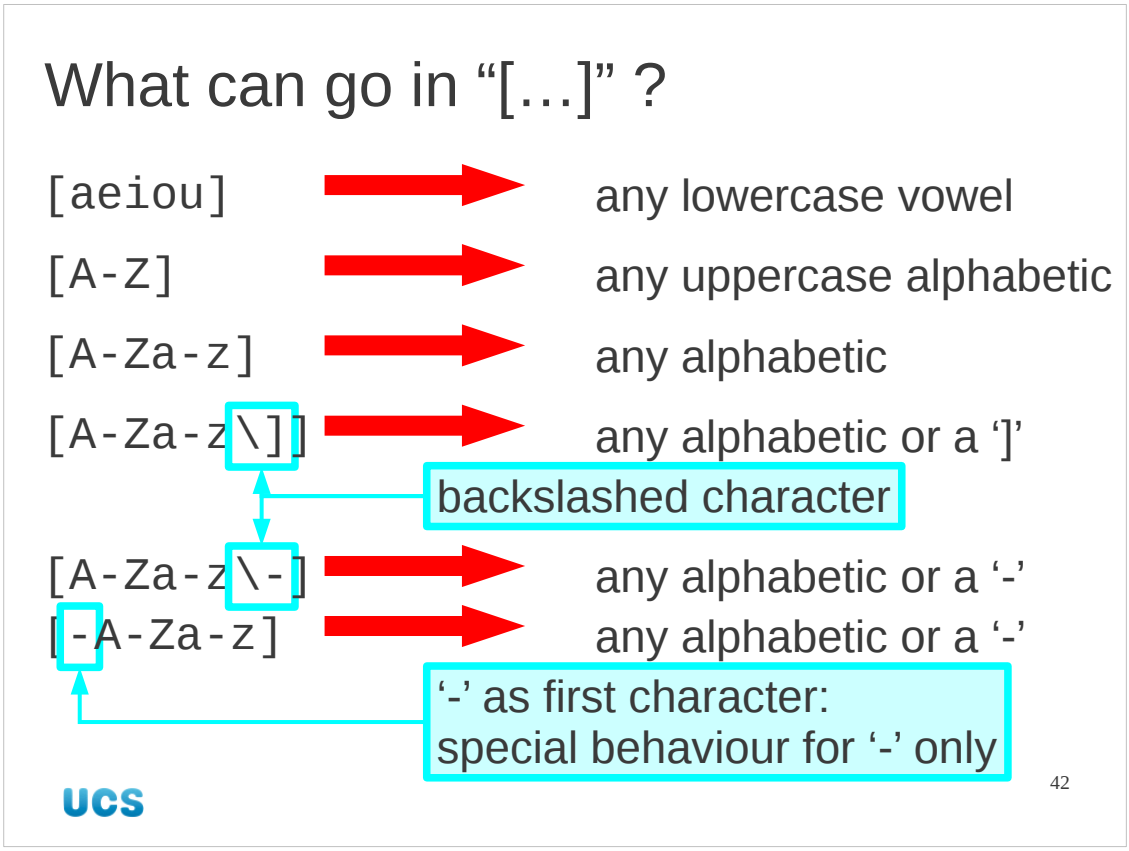
41

We have now started to meet some of the special codes that the regular expression language uses in its patterns.

The caret character, "^", means "start of line". The caret is traditional, but there is an equivalent which is "\A".

The dollar character, "$", means "end of line" and has a "\Z" equivalent.

The sequence "\d" means "a digit". Note that the capital version, "\D" means exactly the opposite.



We also need to consider just what can go in between the 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".

What can go in “[…]” ?

`[^aeiou]`	→	*not* any lowercase vowel
`[^A-Z]`	→	*not* any uppercase alphabetic
`[\^A-Z]`	→	any uppercase alphabetic or a caret

UCS

If the first character in the square brackets is a caret (“^”) then the sense of the term is reversed; it stands for any one character that is not one of those in the square brackets.

If you want to have a true caret in the set, precede it with a backslash.

Counting in regular expressions

`[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’.

UCS

44

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

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”.

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.

The more precise counting is done wit curly brackets.

What matches “[” ?

“[abcd]” matches any one of “a”, “b”, “c” or “d”.

What matches “[abcd]”?

`[abcd]` → Any one of ‘a’, ‘b’, ‘c’, ‘d’.

`\[abcd\]` → `[abcd]`

UCS

45

Now let's pick up a few stray questions that might have arisen as we built that pattern.

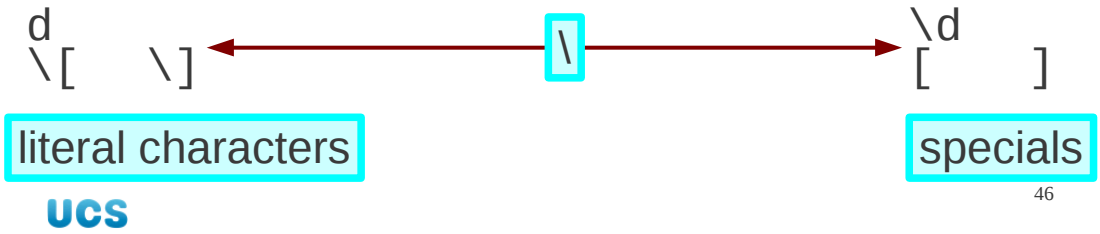
If square brackets identify sets of letters to match, what matches a square bracket? How would I match the *literal* string “`[abcde]`”, for example?

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 “`\\`”.)

We will see more about backslash next.

Backslash

`[ ]`	used to hold sets of characters
`\[ \]`	the real square brackets
`d`	the letter “d”
`\d`	any digit



UCS

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 “`\\`”.)

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” but “`\d`” matches any one digit.

What does dot match?

We've been using dot as a literal character.

Actually…

`.`	"`.`" matches any character except "`\n`".
`\.`	"`\.`" matches just the dot.

UCS

47

There's also an issue with using "`.`". We've been using it as a literal character, matching the full stops at the ends of sentences, or in file name suffixes but actually it's another special character that matches any single character except for the new line character ("`\n`" matches the new line character). We've just been lucky so far that the only possible match has been to a real dot. If we want to force the literal character we place a backslash in front of it, just as we did with square brackets.

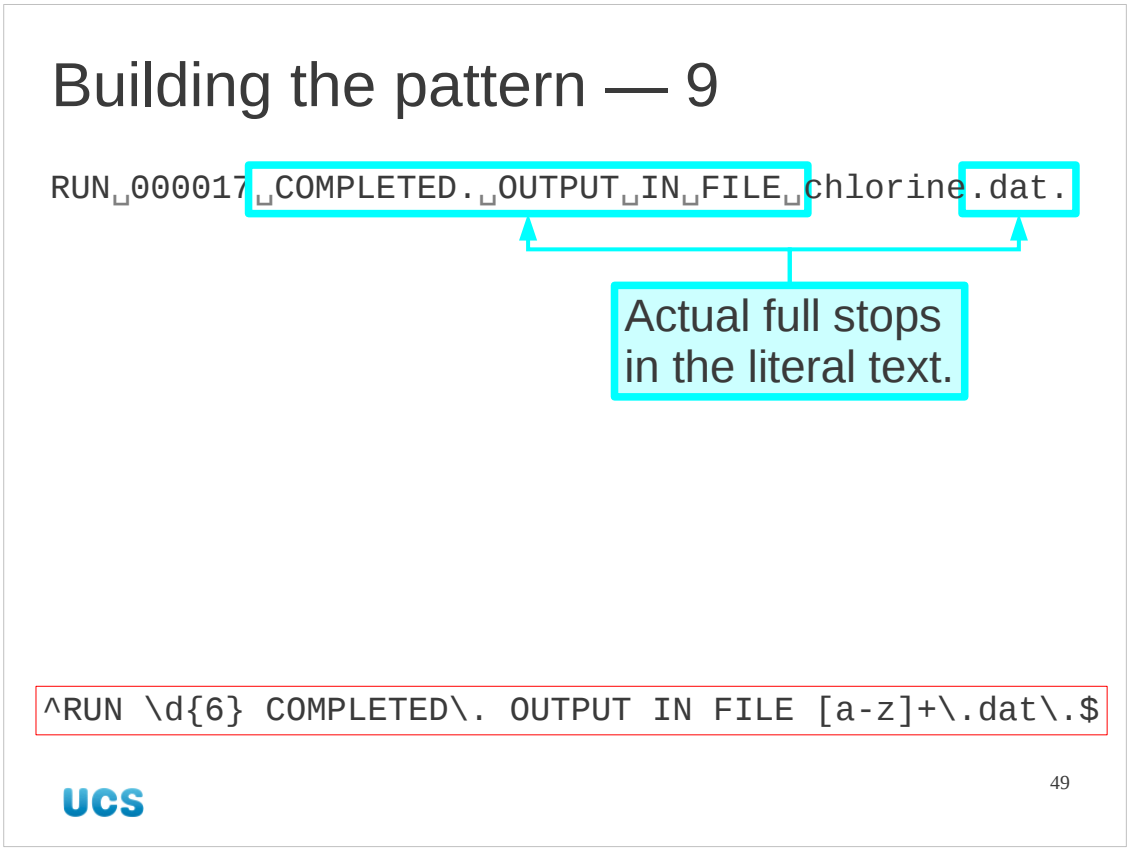
Special codes in regular expressions

\A	**^**	Anchor start of line
\Z	**$**	Anchor end of line
\d		Any **d**igit
\D		Any non-digit
.		Any character except newline

UCS

48

So we can add the full stop to our set of special codes.



So our filter expression for the `atoms.log` file needs a small tweak to indicate that the dots are real dots and not just “any character except the newline” markers.

Exercise 4: changing the atom filter

1. Edit `filter03.py`

   Fix the dots.

2. Run `filter03.py` again to check it.

   ```
   $ python filter03.py < atoms.log
   ```

UCS

50

So apply this change to the dots to your script that filters the atoms log.

Exercise 5 and coffee break

Input: `messages`

Script: `filter04.py`

Match lines with "Invalid user".

Match the *whole* line.

"Grow" the pattern one bit at a time.

15 mins

UCS

51

We'll take a break to grab some coffee. Over the break try this exercise. Copy the file "`filter03.py`" to "`filter04.py`" and change the pattern to solve this problem:

The file "`messages`" contains a week's logs from one of the authors' workstations. In it are a number of lines containing the phrase "`Invalid user`". Write a regular expression to match these lines and then print them out.

Match the whole line, not just the phrase. We will want to use the rest of the line for a later exercise. In addition, it forces you to think about how to match the terms that appear in that line such as dates and time stamps.

This is a complex pattern. We strongly suggest building the pattern one bit at a time. Start with "`^[A-Z][a-z]{2}␣`" to match the month at the start of the line. Get that working. Then add the day of the month. Get that working. Then add the next bit and so on.

There are 1,366 matching lines. Obviously, that's too many lines for you to sensibly count just by looking at the screen, so you can use the Unix command `wc` to do this for you like this:

```
$ python filter04.py < messages | wc -l
1366
```

(The "`-l`" option to `wc` tells it just to count the number of lines in the output. The pipe symbol, "`|`", tells the operating system to take the output of your Python script and give it to the `wc` command as input.)

Answer to exercise

```
^[A-Z][a-z]{2}␣[123␣][0-9]␣\d\d:\d\d:\d\d␣
noether␣sshd\[\d+\]:␣Invalid␣user␣[A-Za-z0-9/\-]+␣
from␣\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$
```

`^`	Start of line
`[A-Z][a-z]{2}`	“Jan”, “Feb”, “Mar”, …
`[123␣][0-9]`	“␣2”, “12”, …
`\d\d:\d\d:\d\d`	“01:23:34”, “12:34:50”, …
`\[\d+\]`	“[567]”, “[12345]”
`[A-Za-z0-9/\-]+`	“admin”, “www-data”, “mp3”, …
`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`	“131.111.4.12”, …
`$`	End of line

UCS

52

Here’s the answer to the exercise. If any of it doesn't make sense, please tell the lecturer now.

The patterns aren’t perfect. The pattern used to match the three character abbreviation of the month also matches “`Jeb`” and “`Fan`”, for example.

Note how to match dates which run from 1 to 31 we don’t say “two digits”. We say “a leading 1, 2, 3 or space” followed by an arbitrary digit. Again, this matches some things which are not dates, 32 for example, but does exclude some other not-dates.

The pattern “`[A-Za-z0-9/\-]+`” to match the IDs being offered is based purely on visual inspection of those in the file. In the future a hacker may attempt to access the system using a login with a different character in it. We will see how to deal with this uncertainty very soon.

Tightening up the regular expression

`\s+`	general white space
`\S+`	general non-white space
`\`	Backslash is special in Python strings (“\n”)
`r'…'`	Putting an “r” in front of a string turns off any special treatment of backslash. (Routinely used for **r**egular expressions.)
`r"…"`	

UCS

53

At this point we ought to develop some more syntax and a useful Python trick to help us round some problems which we have skated over.

An issue with breaking up lines like this is that some systems use single spaces while others use double spaces at the ends of sentences or tab stops between fields etc. The sequence “`\s`” means “a white space character” (space and tab mostly) so “`\s+`” is commonly used for “some white space”.

Note that white space is marked by a backslashed lower case “`s`”. An upper case “S” means exactly the opposite. “`\s`” matches a single white space character and “`\S`” matches a single character that is *not* white space. This will let us work round the problem of not knowing what characters will appear in the invalid logins in advance.

We seem to be using backslash a lot in regular expressions. Unfortunately backslash is also special for ordinary Python strings. “`\n`”, for example means a new line, “`\t`” means a tab, and so on. We want to make sure that our regular expression use of backslash does not clash with the Python use of backslash. The way we do this is to precede the string with the letter “`r`”. This turns off any special backslash handling Python would otherwise do. This is usually only done for regular expressions.

The final pattern

```
^[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}$
```

UCS

So our exercise's pattern gets this final form.
We have replaces the "`[A-Za-z0-9/\-]+`" pattern which happened to work for our particular log file with "`\S+`" (that's an *uppercase* "S") to match more generally.

The final string

```
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}$'
```

UCS

55

And the corresponding Python string in the script gets a leading “r” to force “raw mode” to stop any Pythonic interpretation of the backslashes, leaving them for the regular expression system.

Special codes in regular expressions

\A	**^**	Anchor start of line
\Z	**$**	Anchor end of line
\d		Any **d**igit
\D		Any non-digit
.		Any character except newline
\s		Any white-**s**pace
\S		Any non-white-space
\w		Any **w**ord character (letter, digit, "_")
\W		Any non-word character

UCS

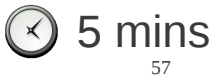
So we can add `\s` and `\S` to our set of special codes and we will take the opportunity to include just two more. The code `\w` matches any character that's likely to be in a (computerish) word. These are the letters (both upper and lower case), the digits and the underscore character. There is no punctuation included. The upper case version means the opposite again.

Exercise 5: improving the filter

Copy `filter04.py` → `filter05.py`

`[A-Za-z0-9/\-]+` → `\S+`

`"…"` → `r"…"`



5 mins

57

UCS

Now you can improve your `filter04.py` script. You have now done all that can really be done to the pattern to make it as good as it gets.

Example

Find the lines containing

“Jan” or

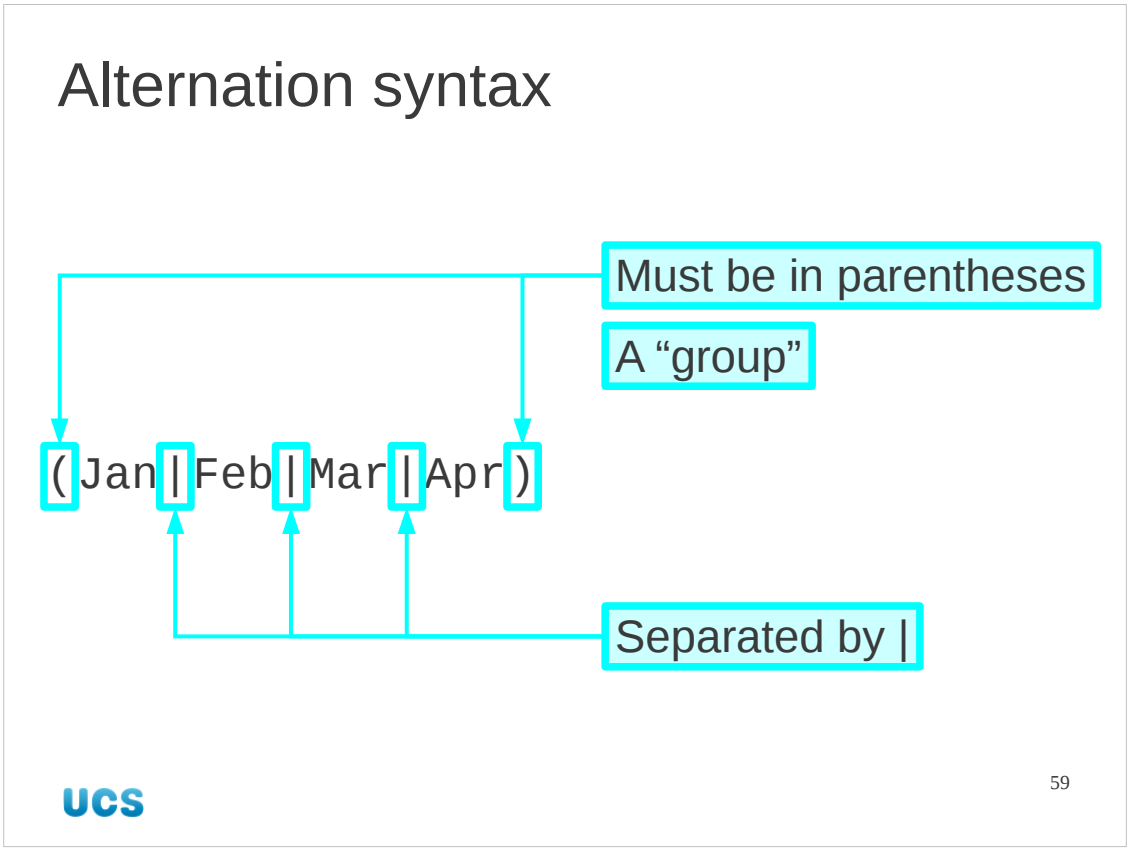
“Feb” or

“Mar” or

“Apr” …

UCS

Our regular expression isn't perfect. Any month abbreviation that starts with an upper case letter followed by two lower case letters will pass. Suppose we really wanted to say “Jan” or “Feb” or “mar” etc.



The syntax we use to describe this “alternation” is as ffollows:

We take the expressions the “month” must match and place them between vertical bars (which we pronounce as “or”) and place the whole thing in parentheses (round brackets).

Any element placed within round brackets is called a “group” and we will be meeting groups in their own right later.

Parentheses in regular expressions

Use a group for alternation	`(…|…|…)`
Use backslashes for literal parentheses	`\(  \)`
Backslash not needed in […]	`[a-z()]`

UCS

We will meet parentheses (and groups) a lot in this course so we will start a slide to keep track of their various uses.

Complex regular expressions

```
^[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}$
```

If regular expressions were
a programming language…

comments

layout

meaningful variable names

UCS

61

As we've just seen, regular expressions can get really complicated. 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.)

Verbose mode

```
^[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}$
```

Problems	Solutions
Hard to write	Multi-line layout
Harder to read	Comments
Hardest to maintain	

UCS

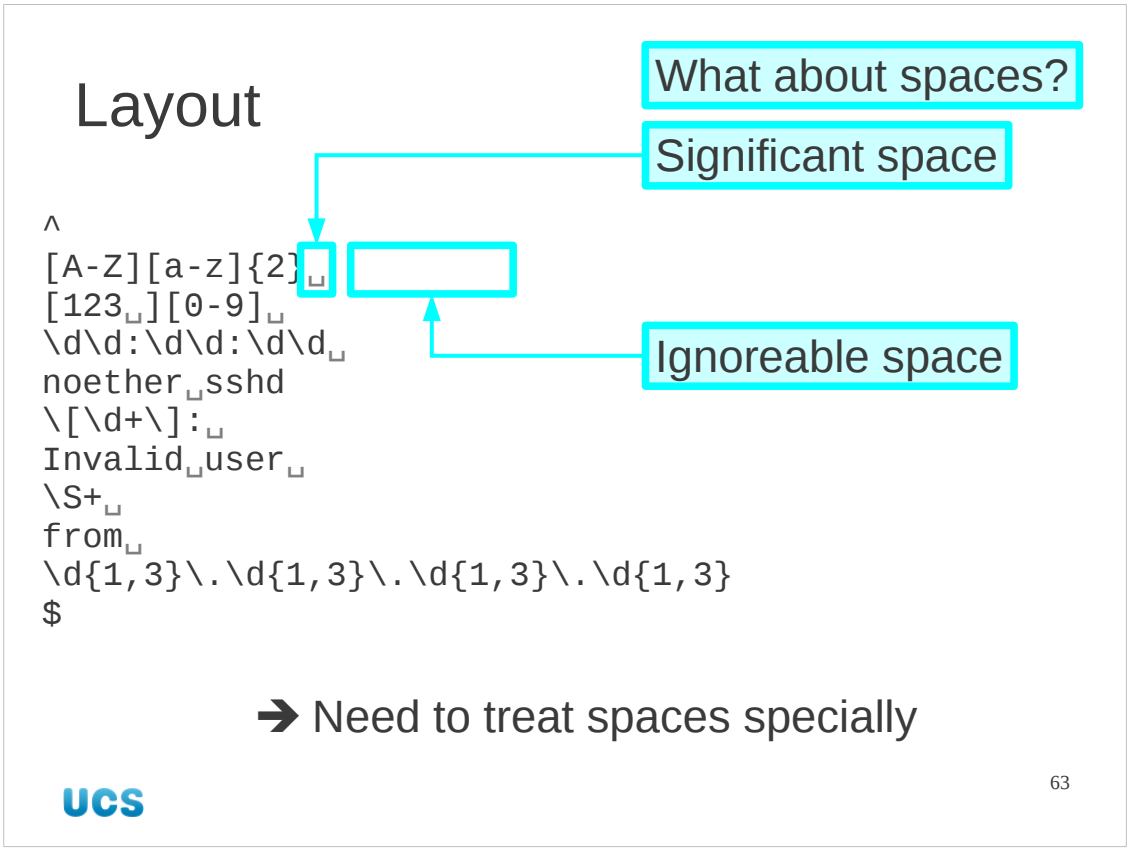
62

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.

The problem is that there is nothing that looks like a useful language for our eyes to hook on; it looks too much like nonsense.

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”.

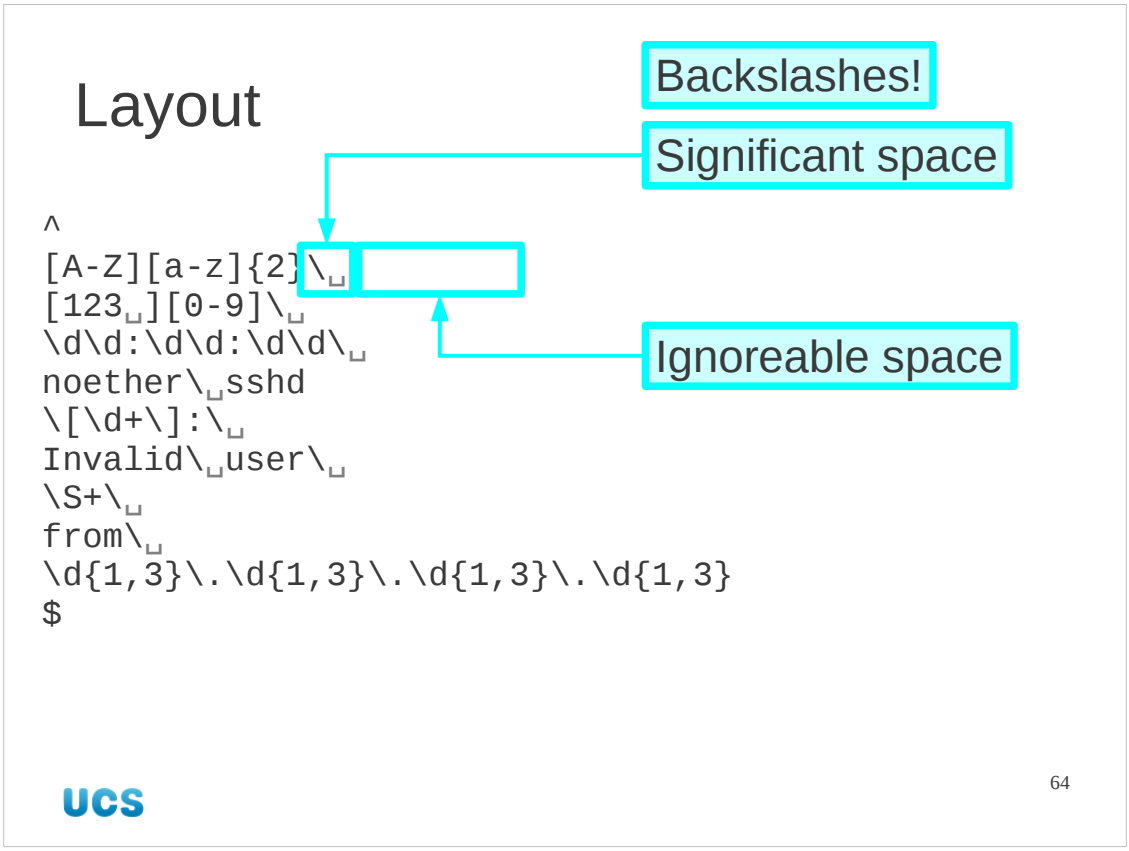


Don't change your script yet!

To fix our layout concerns let's try splitting our pattern over several lines. We'll use the one from the exercise as it is by far the most complex pattern we have seen to date.

The first issue we hit concerns spaces. We want to match on spaces, and our original regular expression had spaces in it. However, multi-line expressions like this typically have trailing spaces at the ends of lines ignored. In particular 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.



We use a backslash.

In a verbose regular expression pattern spaces become special characters; “special” because they are completely ignored. So we make a particular space “ordinary” (i.e. just a space) by preceding it with a backslash, just as we did for square brackets.

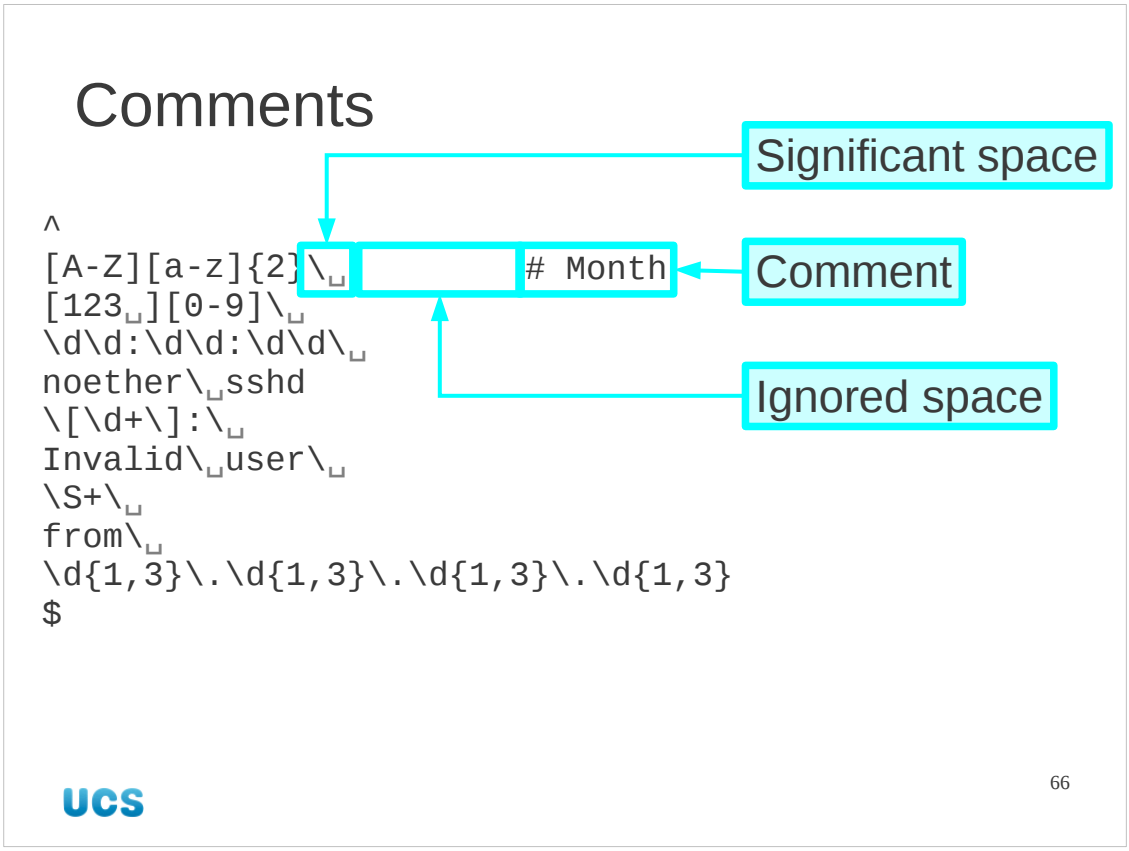
Slightly paradoxically, where the space appears inside square brackets to indicate membership of a set of characters it doesn't need backslashing as its meaning is unambiguous.

Spaces in verbose mode

Space ignored generally	␣
Backslash space recognised	\␣
Backslash not needed in […]	[123␣]

UCS

So this will be a slight change needed to our regular expression language to support multi-line regular expressions.



Now let's add comments.

We will introduce them using exactly the same character as is used in Python proper, the “hash” character, “#”.

Any text from the hash character to the end of the line is ignored.

This means that we will have to have some special treatment for hashes if we want to match them as ordinary characters, of course. It's time for another backslash.

Hashes in verbose mode

Hash introduces a comment	`# Month`
Backslash hash matches “#”	`\#`
Backslash not needed in […]	`[123#]`

UCS

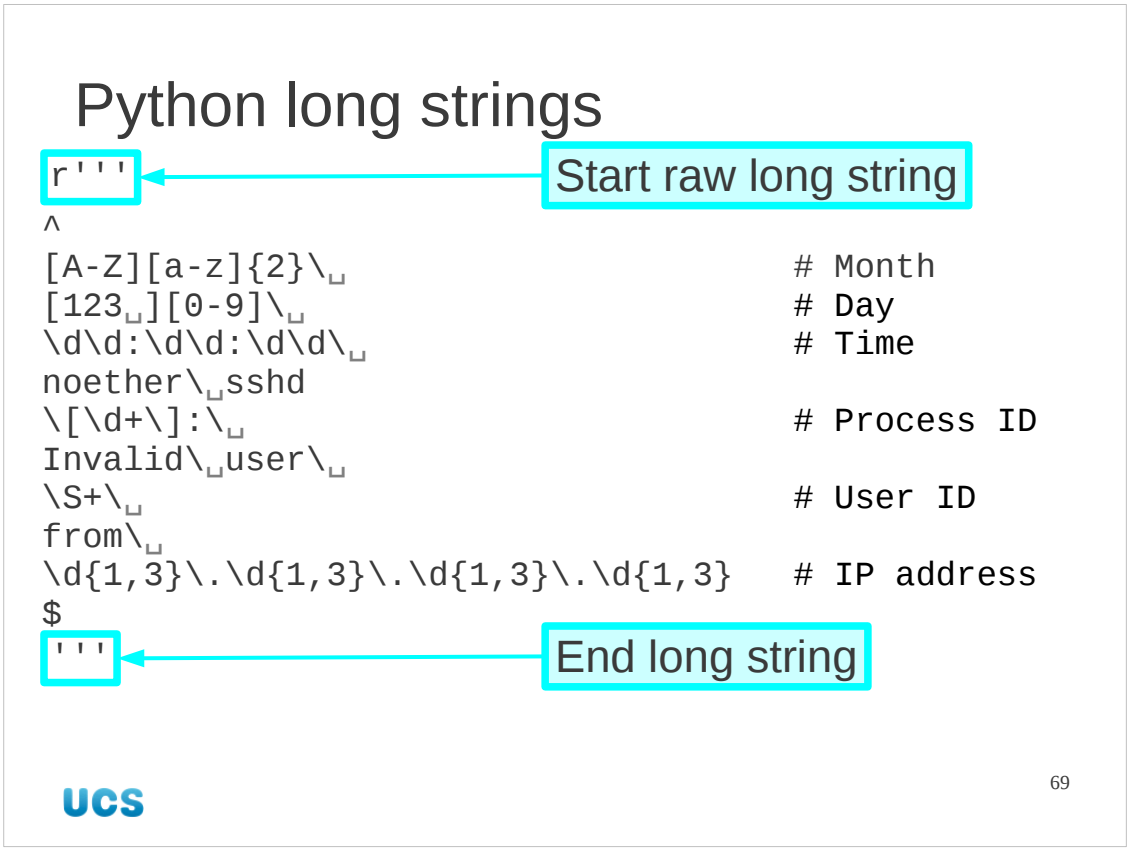
In multi-line mode, hashes introduce comments. The backslashed hash, “`\#`”, matches the hash character itself. Again, just as with space, you don't need the backslash inside square brackets.

Verbose mode

```
^
[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+\␣                                    # User ID
from\␣
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}       # IP address
$
```

UCS

So this gives us our more legible mode. Each element of the regular expression gets a line to itself so it at least looks like smaller pieces of gibberish. Furthermore each can have a comment so we can be reminded of what the fragment is *trying* to match.



This verbose regular expression pattern covers many lines. Python has a mechanism specifically designed for multi-line strings: the triple-quoted string. We always use that, in conjunction with the `r` (“**r**aw”) qualifier to carry these verbose regular expression patterns.

Telling Python to “go verbose”

Verbose mode

Another option, like ignoring case

Module constant, like `re.IGNORECASE`

`re.VERBOSE`
`re.X`

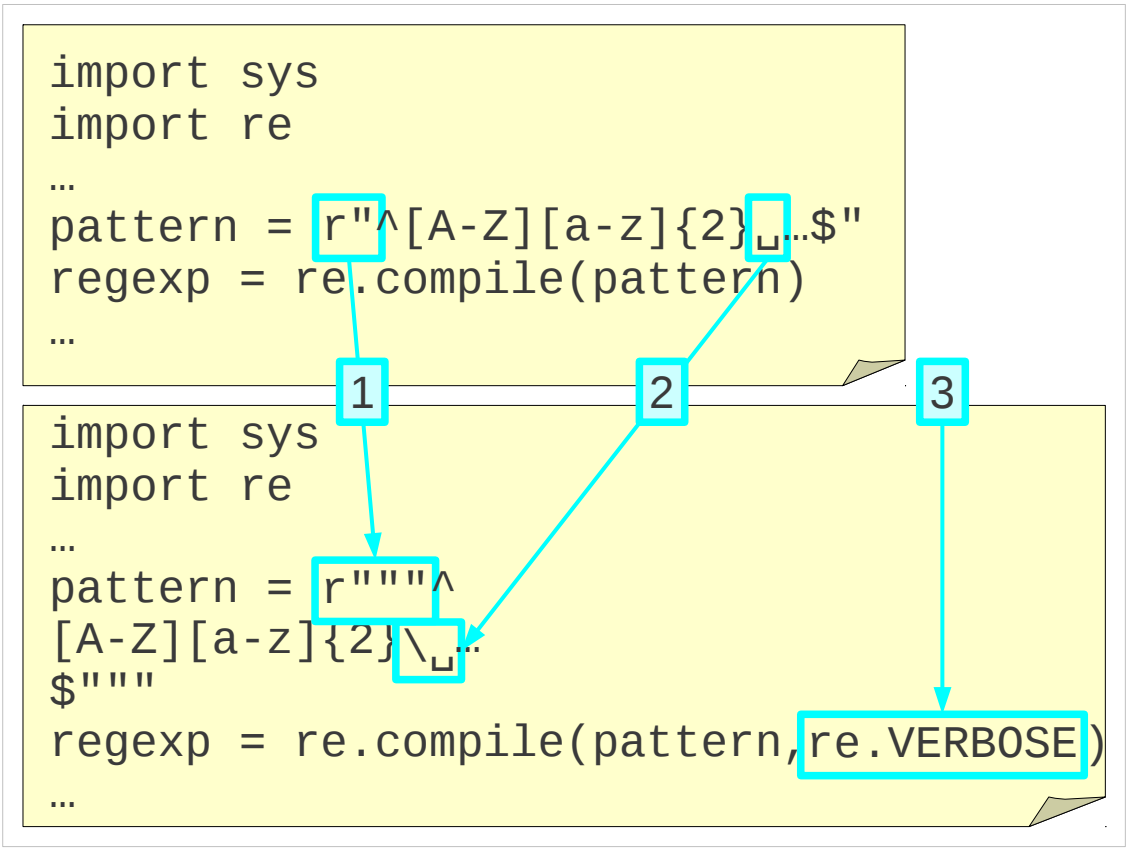
UCS

70

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 chort 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 three 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 7: use verbose mode

10 mins

72

UCS

So now that you've seen how to turn an "ordinary" regular expression into a "verbose" one with comments, it's time to try it for real.

Copy the files `filter03.py` and `filter04.py` and edit the copies so that the regular expression patterns they use are "verbose" ones laid out across multiple lines with suitable comments. Test them against the same input files as before.

```
$ cp filter03.py filter05.py
$ gedit filter05.py
$ python filter05.py < atoms.log

$ cp filter04.py filter06.py
$ gedit filter06.py
$ python filter06.py < messages
```

As ever, if you have any problems with this exercise, please ask the lecturer.

In each edit you will need to convert the pattern and set the compilation option.

Extracting bits from the line

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

Suppose we wanted to extract just these two components.

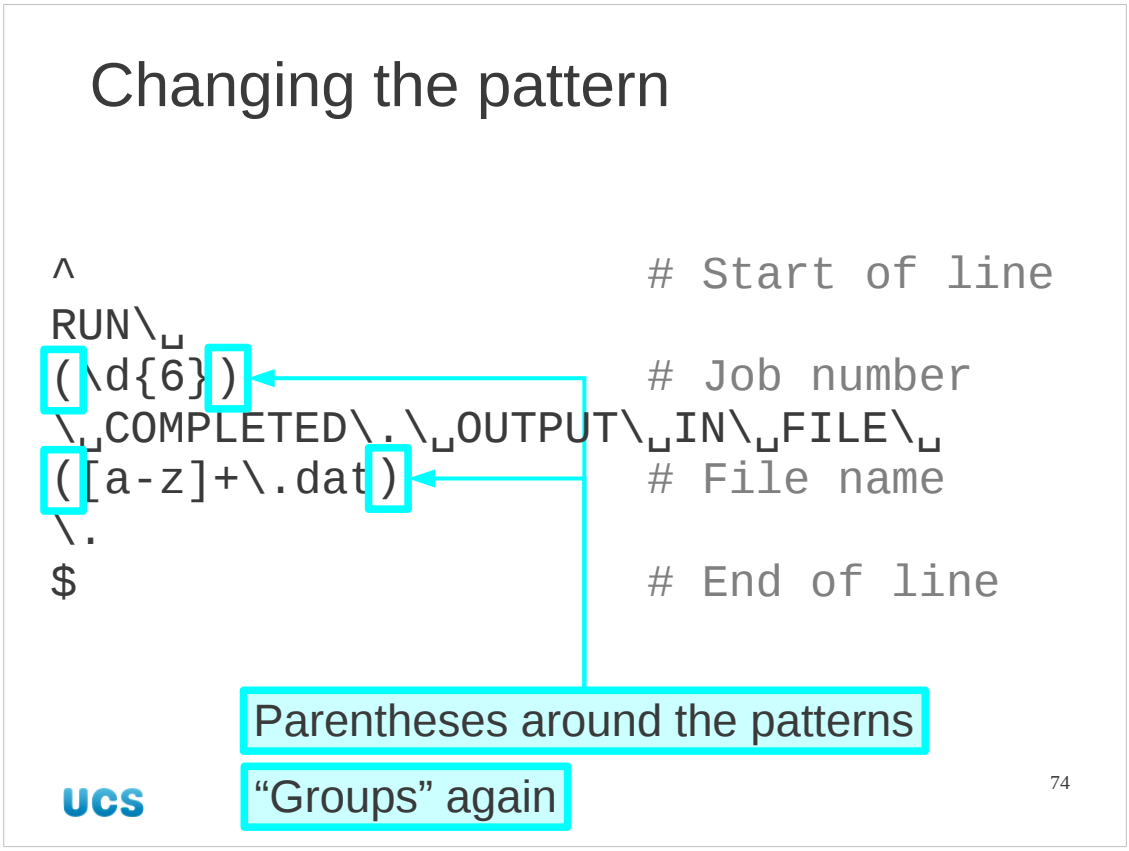
UCS

73

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! Note that nothing we teach in the remainder of this course is specific to verbose mode; it will all work equally well in the concise mode too.

Suppose we are particularly interested in two parts of the line, the job number and the file name. Note that the file name includes both the component that varies from line to line, "`[a-z]+`", and the constant, fixed suffix, "`.dat`".

What we will do is 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.

Recall the “(Jan|Feb|Mar|Apr)” example. These are groups again, but this time they are groups of just one pattern rather than a chain of them.

The “match object”

```
…
regexp = re.compile(pattern,re.VERBOSE)

for line in sys.stdin:
    result = regexp.search(line)
    if result:
        …
```

UCS

75

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.

Using the match object

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

`result.group(1)`	`'000001'`
`result.group(2)`	`'hydrogen.dat'`
`result.group(0)`	*whole pattern*

UCS

We get at the groups from the match object. The method `result.group(1)` will return the contents of the first pair of parentheses and the method `result.group(2)` will return the content of the second.

Avid Pythonistas will recall that Python usually counts from zero and may wonder what `result.group(0)` gives. This returns whatever the entire pattern matched. In our case where our regular expression defines the whole line (^ to $) this is equivalent to the whole line.

Putting it all together

```
…
regexp = re.compile(pattern,re.VERBOSE)

for line in sys.stdin:
    result = regexp.search(line)
    if result:
        sys.stdout.write("%s\t%s\n" % (r
esult.group(1), result.group(2)))
```


UCS

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

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.

Parentheses in regular expressions

Use a group for alternation	`(…|…|…)`
Use backslashes for literal parentheses	`\(  \)`
Backslash not needed in […]	`[a-z()]`
Use a group for selection	`(…)`

UCS

78

If you want to match a literal parenthesis use "`\(`" or "`\)`".

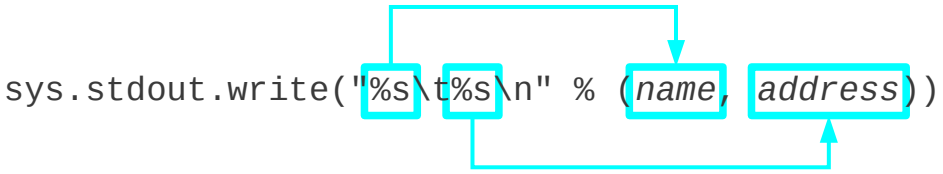
Note that because (unbackslashed) parentheses have this special meaning of defining subsets of the matching line they *must* match. If they don't then the `re.compile()` function will give an error similar to this:

```
>>> pattern='('
>>> regexp=re.compile(pattern)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python2.6/re.py", line 188, in compile
    return _compile(pattern, flags)
  File "/usr/lib64/python2.6/re.py", line 243, in _compile
    raise error, v # invalid expression
sre_constants.error: unbalanced parenthesis
>>>
```

Exercise 8: limited output

Modify the log file filter to output just
the account name and the IP address.

`filter06.py` → `filter07.py`



5 mins

79

UCS

Now try it for yourselves:

You have a file `filter06.py` which you created to answer an earlier exercise. This finds the lines from the messages file which indicate an Invalid user.

Copy this script to `filter07.py`.

Edit `filter07.py` so that you define groups for the account name (matched by `\S+`) and the IP address (matched by `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`).

```
$ cp filter06.py filter07.py
$ gedit filter07.py
$ python filter07.py < messages
```

The bottom of the slide is a quick reminder of the string substitution syntax in Python. This will get you nicely tab-aligned text.

Limitations of numbered groups

The problem:

Insert a group ⟶ All following numbers change

“What was group number three again?”

The solution: use names instead of numbers

Insert a group ⟶ It gets its own name

Use sensible names.

UCS

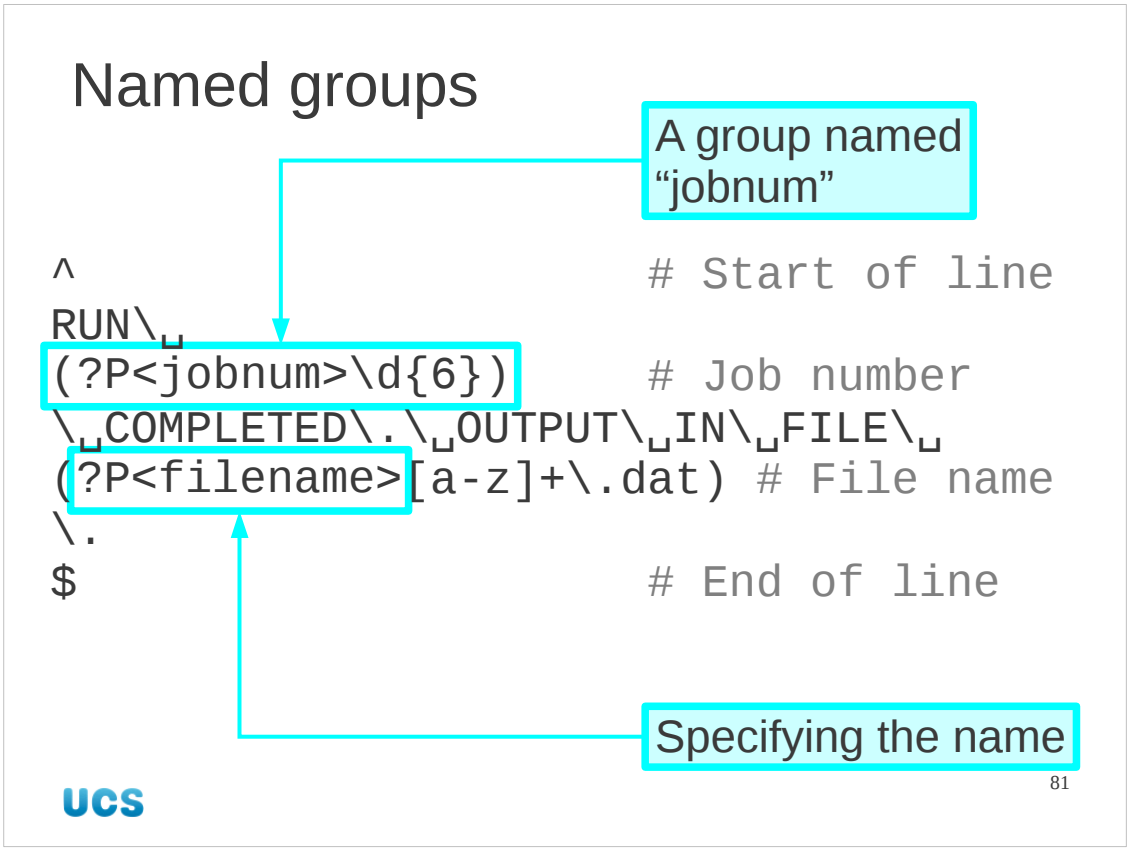
80

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.

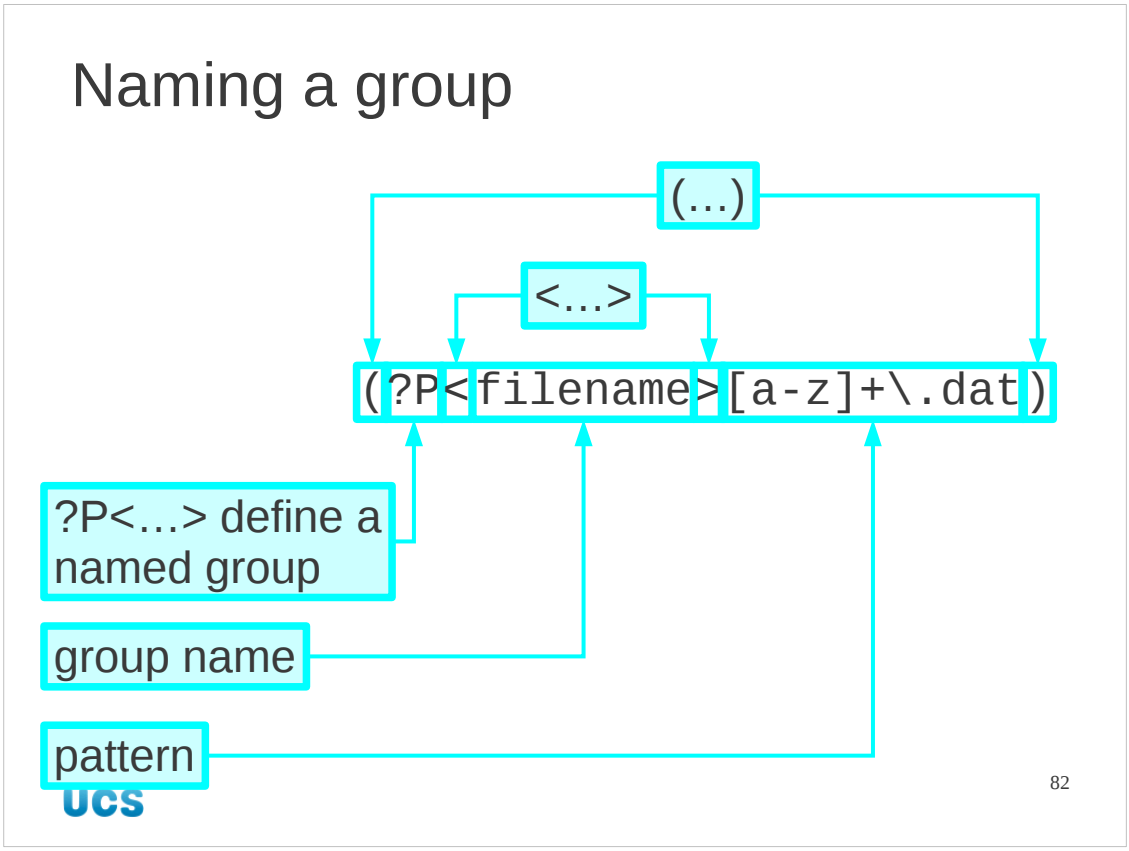
There's also a problem that “`regexp.group(2)`” doesn't shout out what the second group actually was.

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



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>".



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.

Using the named group

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


result.group('jobnum')    '000001'
result.group('filename')  'hydrogen.dat'
```

UCS

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`".

Putting it all together — 1

```
…
pattern=r'''
^
RUN\␣
(?P<jobnum>\d{6})           # Job number
\␣COMPLETED\.\␣OUTPUT\␣IN\␣FILE\␣
(?P<filename>[a-z]+\.dat)  # File name
\.
$
'''

…
```


UCS

So if we edit our `filter05.py` script we can allocate group name “`jobnum`” to the series of six digits and “`filename`” to the file name (complete with suffix “`.dat`”). This is all done in the pattern string.

Putting it all together — 2

```
…
regexp = re.compile(pattern,re.VERBOSE)

for line in sys.stdin:
    result = regexp.search(line)
    if result:
        sys.stdout.write("%s\t%s\n" % r
esult.group('jobnum'), result.group('fi
lename'))
```


UCS

At the bottom of the script we then modify the output line to use the names of the groups in the write statement.

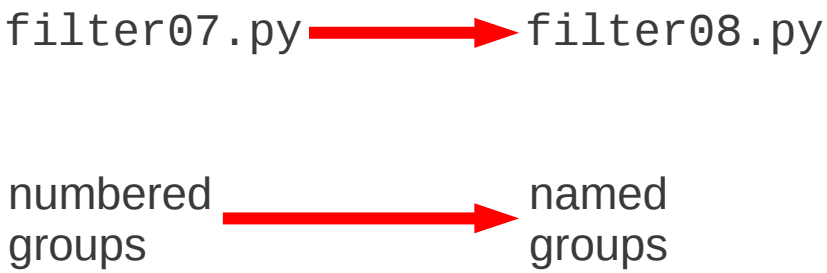
Parentheses in regular expressions

Alternation	`(…|…|…)`
Backslashes for literal parentheses	`\(  \)`
Backslash not needed in […]	`[a-z()]`
Numbered selection	`(…)`
Named selection	`(?P<name>…)`

UCS

So here’s a new use of parentheses: named groups.

Exercise 9: used named groups



5 mins

87

UCS

Now try it for yourselves. Make a copy of the `filter07.py` script in `filter08.py` and edit the copy to use named groups (with meaningful group names). Make sure you test it to check it still works!

If you have any problems with this exercise, please ask the lecturer.

Ambiguous groups within the regular expression

Dictionary: `/var/lib/dict/words`

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

Script: `filter09.py`

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

UCS

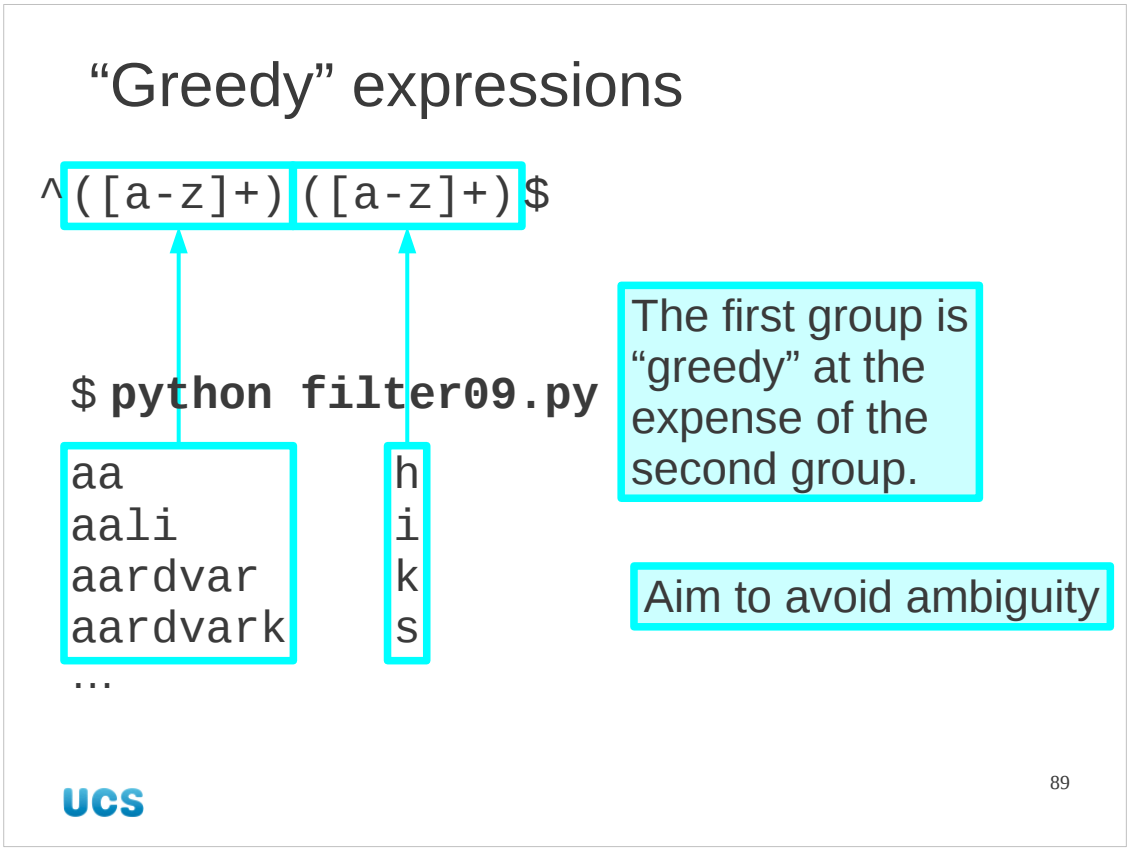
88

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 `/var/lib/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 `filter09.py` which is currently in your home directory:

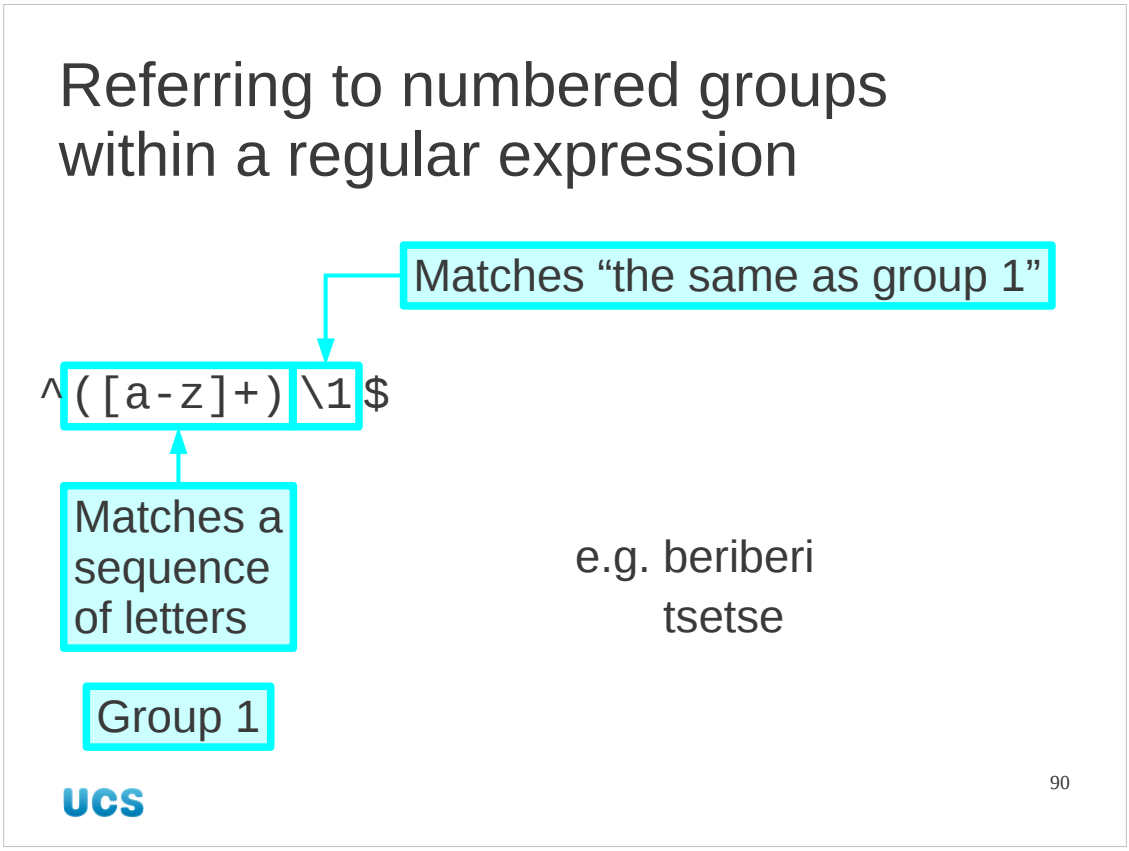
```
$ python filter09.py
aa  h
aahed
aahin   g
aah s
aa  l
aalii
aalii   s
aal s
aardvar  k
...
```



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.

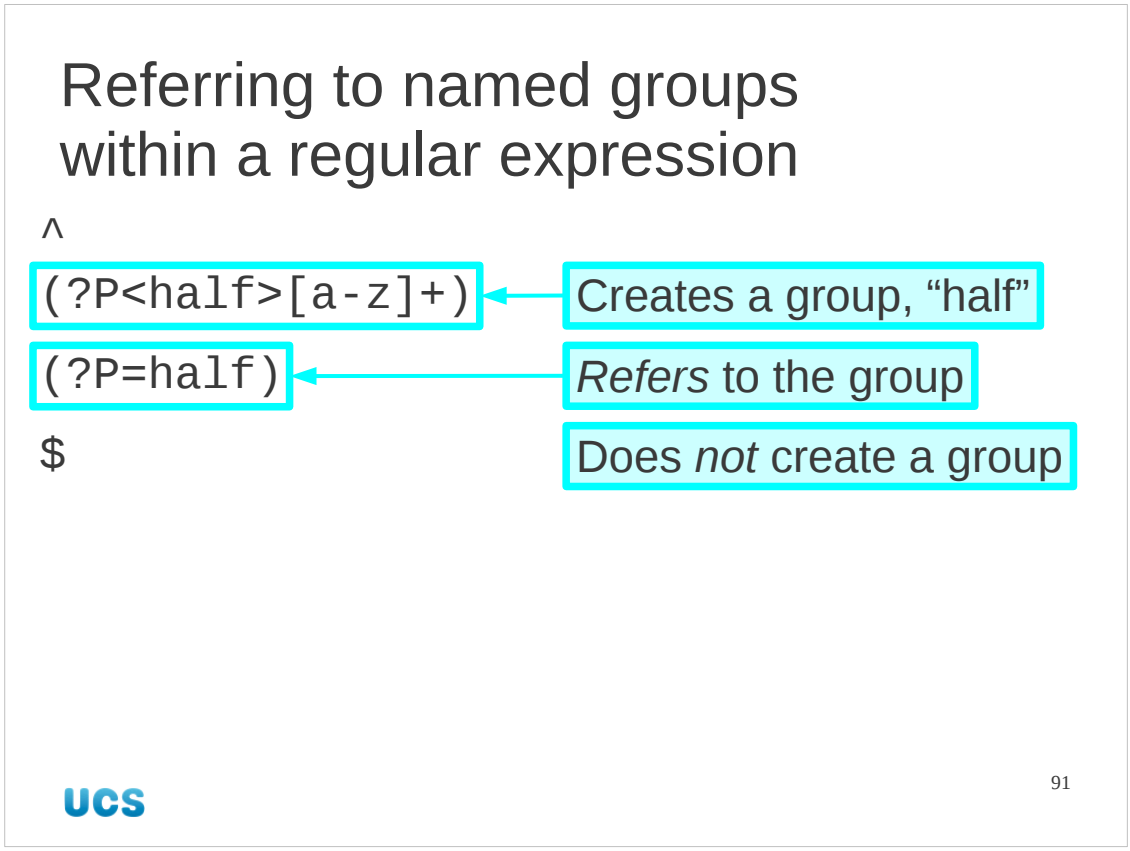
You can change the greed of various groups with yet more use of the query character but please note the ambiguity caution above. If you find yourself wanting to play with the greediness you're almost certainly doing something wrong at a deeper level.



In our ambiguity example, `filter09.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.



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<word>[a-z]+)(?P=word)$`” 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.

Example

```
$ python filter10.py
atlatl
baba
beriberi
bonbon
booboo
bulbul
...
```

UCS

The file `filter06.py` does precisely this using a named group.
I have no idea what half of these words mean.

Parentheses in regular expressions

Alternation	`(…|…|…)`
Backslashes for literal parentheses	`\(  \)`
Backslash not needed in […]	`[a-z()]`
Numbered selection	`(…)`
Named selection	`(?P<name>…)`
Named reference	`(?P=name)`

UCS

This completes the next set of uses of parentheses in Python regular expressions. Remember that the final “reference” example does not create a group.

Exercise 10

`filter10.py` → `filter11.py`

Find all the words with the pattern *A***B***A***B***A*

e.g. entente → entente

A *A* *A* → e

B **B** → nt

UCS

94

Copy the script `filter10.py` to `filter11.py` and edit the latter to find all the words with the form ABABA. (Call your groups “a” and “b” if you are stuck for meaningful names.

Note that in for the example word on the slide, the A pattern just happens to be one letter long (the lower case letter “e”), whilst the B pattern is two letters long (the lower case letter sequence “nt”).

Hint: On PWF Linux the `/var/lib/dict/words` dictionary contains 5 such words. No, I have no idea what most of them mean, either.

Multiple matches

Data: `boil.txt`

Basic entry: `Ar 87.3`

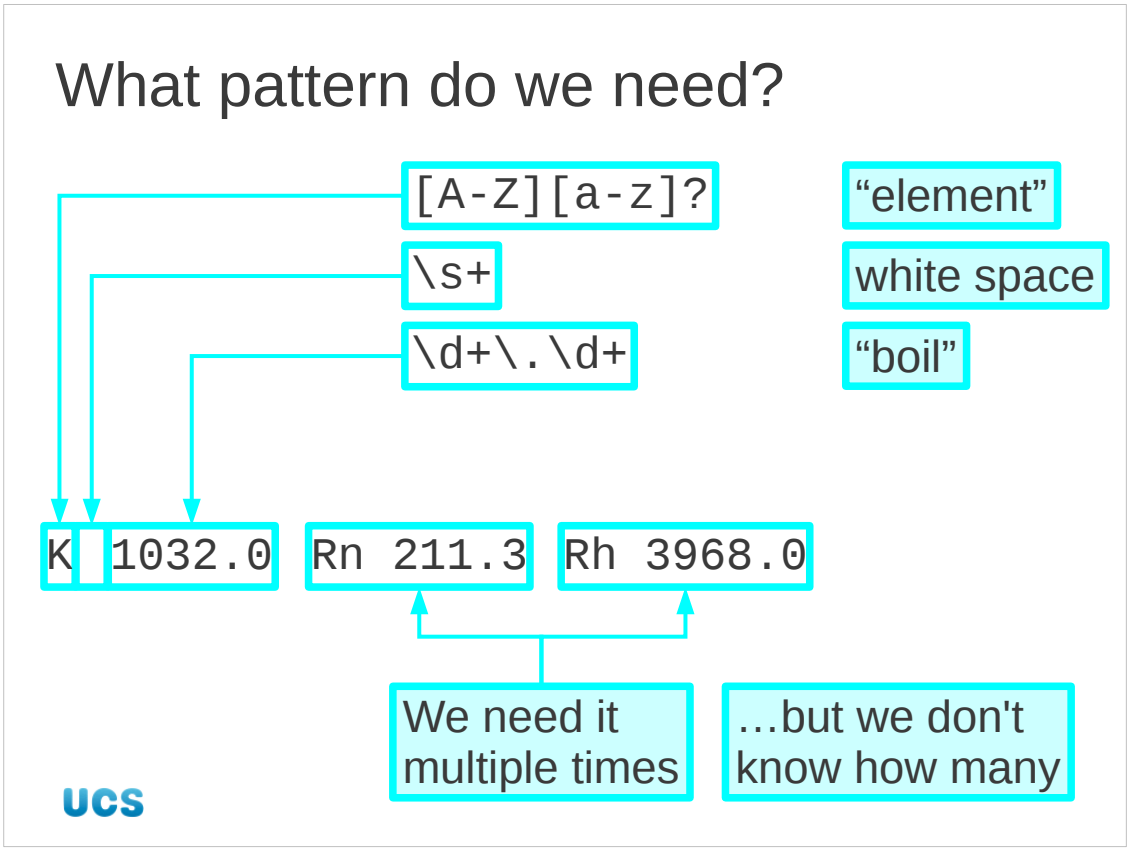
Different number of entries on each line:

```
Ar 87.3
Re 5900.0 Ra 2010.0
K 1032.0 Rn 211.3 Rh 3968.0
```

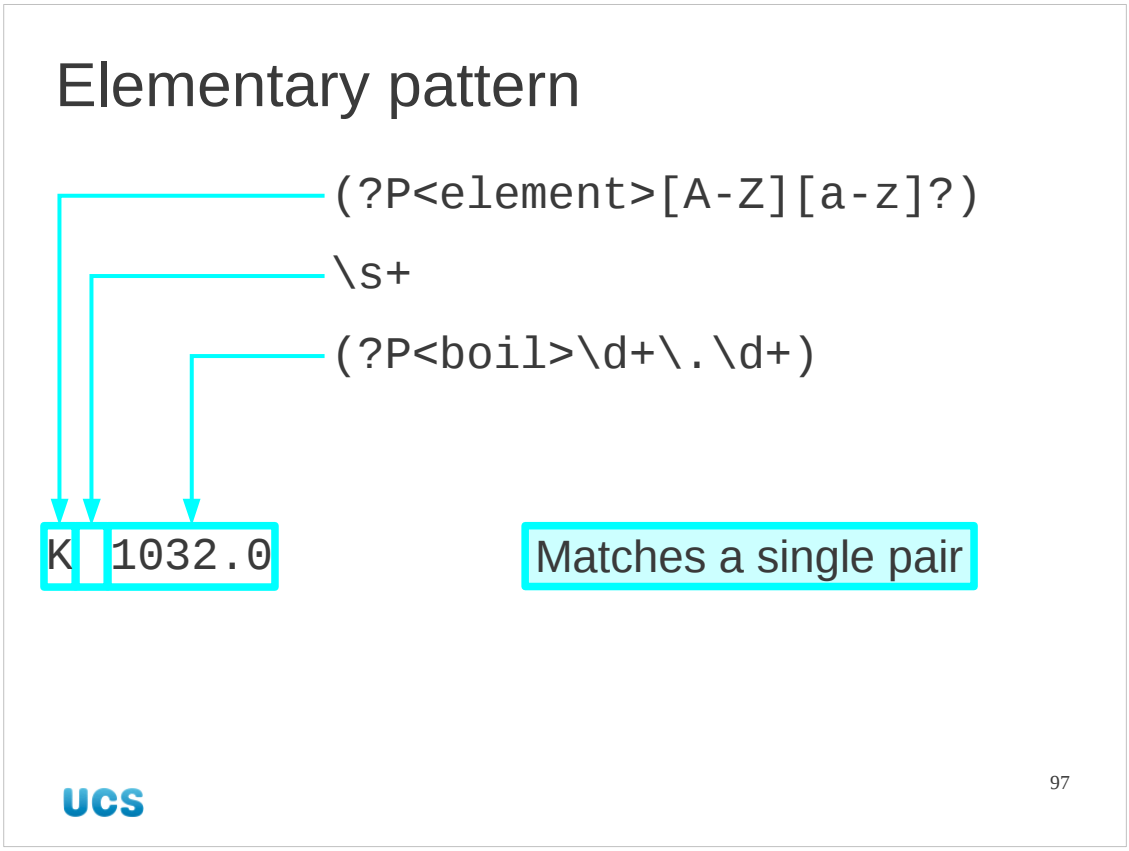
Want to unpick this mess

UCS

Now we will move on to a more powerful use of groups. Consider the file `boil.txt`. This contains the boiling points (in Kelvin at standard pressure) of the various elements but it has lines with different numbers of entries on them. Some lines have a single element/temperature pair, others have two, three, or four. We will presume that we don't know what the maximum per line is.



The basic structure of each line is straightforward, so long as we can have an arbitrary number of instances of a group.



We start by building the basic pattern that will be repeated multiple times. The basic pattern contains two groups which isolate the components we want from each repeat: the name of the element and the temperature.

Note that because the pattern can occur anywhere in the line we don't use the “^/$” anchors.

Putting it all together — 1

```
…
pattern=r'''
(?P<element>[A-Z][a-z]?)
\s+
(?P<boil>\d+\.\d+)
'''

regexp = re.compile(pattern,re.VERBOSE)
…
```


UCS

We put all this together in a file call `filter12.py`.

Putting it all together — 2

```
…

for line in sys.stdin:
    result = regexp.search(line)
    if result:
        sys.stdout.write("%s\t%s\n" % r
esult.group('element'), result.group('b
oil'))
```


UCS

At the bottom of the script we print out whatever the two groups have matched.

Try that pattern

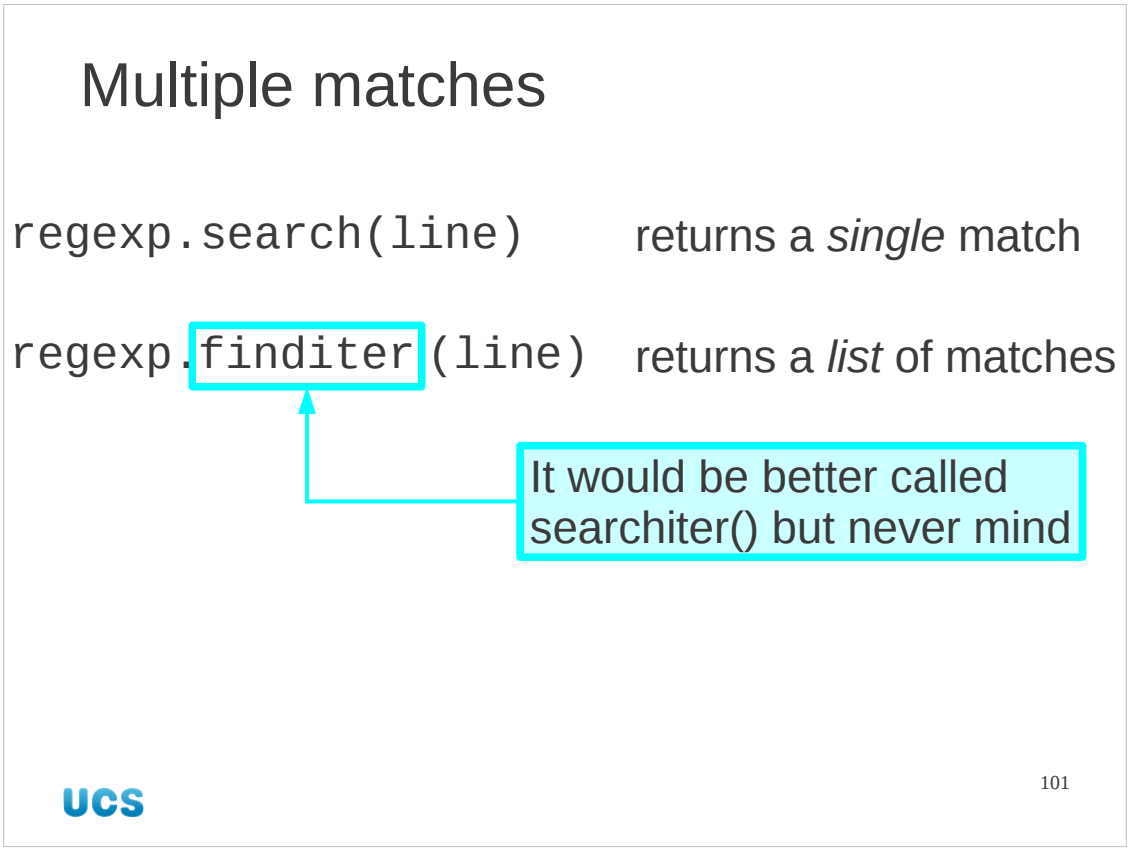
```
$ python filter12.py < boil.txt
Ar  87.3
Re  5900.0
K   1032.0
…
Ag  2435.0
Au  3129.0
```

First matching case of each line

But only the *first*

UCS

We will start by dropping this pattern into our standard script, mostly to see what happens. The script does generate some output, but the pattern only matches against the start of the line. It finishes as soon as it has matched once.



The problem lies in our use of `regexp`'s `search()` method. It returns a single `MatchObject`, corresponding to that first instance of the pattern in the line.

The regular expression object has another method called "`finditer()`" which returns a *list* of matches, one for each that it finds in the line. (It would be better called "`searchiter()`" but never mind.)

(Actually, it doesn't return a list, but rather one of those Python objects that can be treated like a list. They're called "iterators" which is where the name of the method comes from.)

The original script

```
…
for line in sys.stdin:


    result = regexp.search(line)
    if result:


        sys.stdout.write("%s\t%s\n" % r
esult.group('element'), result.group('b
oil'))
```


UCS

So, we return to our script and observe that it currently uses `search()`to return a single `MatchObject` and tests on that object.

The changed script

```
…
for line in sys.stdin:


    results = regexp.finditer(line)
    for result in results:


        sys.stdout.write("%s\t%s\n" % r
esult.group('element'), result.group('b
oil'))
```


UCS

The pattern remains exactly the same.

We change the line that called `search()` and stored a single `MatchObject` for a line that calls `finditer()` and stores a list of `MatchObjects`.

Instead of the `if` statement we have a `for` statement to loop through all of the `MatchObjects` in the list. (If none are found it's an empty list.)

This script can be found in `filter13.py`.

Using finditer()

```
$ python filter13.py < boil.txt
Ar  87.3
Re  5900.0
Ra  2010.0
…
Au  3129.0
At  610.0
In  2345.0
```

Every matching case in each line

UCS

And it works! This time we get all the element/temperature pairs.

Exercise 11

`filter14.py`

Edit the script so that text is split into one word per line with no punctuation or spaces output.

```
$ python filter14.py < paragraph.txt
This
is
free
…
```

UCS

One last exercise in class. The file `filter14.py` that you have is a skeleton script that needs lines completed. Edit the file so that it can be used to split incoming text into individual words, printing one on each line. Punctuation should not be printed. You may find it useful to recall the definition of "`\w`".

We have covered only simple regular expressions!

Capable of much more!

We have focused on getting Python to use them.

UCS course:
“Pattern Matching Using Regular Expressions” focuses on the expressions themselves and not on the language using them.

UCS

106

And that's it!

However, let me remind you that this course has concentrated on getting regular expressions to work in Python and has only introduced regular expression syntax where necessary to illustrate features in Python's `re` module. Regular expressions are capable of much, much more and the UCS offers a two afternoon course, “Pattern Matching Using Regular Expressions”, that covers them in full detail. For further details of this course see the course description at:

`http://training.csx.cam.ac.uk/course/regex`