(modified subset of) Python: Regular Expressions

Originally by: Bruce Beckles, Bob Dowling

University Computing Service

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

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

Classic regular expression filter

for each line in a file : ← Python idiom

does the line match a pattern? ← how can we tell?

if it does, output something ← what?

“Hey! Something matched!”

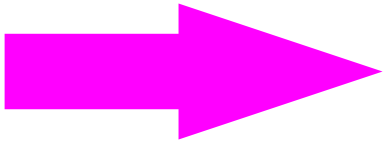
The line that matched

The bit of the line that matched

Task: Look for “Fred” in a list of names

Alice
Bob
Charlotte
Derek
Ermintrude
Fred
Freda
Frederick
Felicity
...

`names.txt`



Fred
Freda
Frederick

`freds.txt`

Skeleton Python script

```
import  regular expression module

define pattern

set up regular expression

for line in open('in.txt'):

    compare line to regular expression

    if  regular expression matches:

        print line
```

read in the lines one at a time

write out the matching lines

Skeleton Python script — 1

```
import re
```

Ready to use regular expressions

define pattern

set up regular expression

```
for line in open('in.txt'):
```

compare line to regular expression

```
    if  regular expression matches:
        print line
```

Skeleton Python script — 2

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

Define the pattern

set up regular expression

```
for line in open('in.txt'):
```

compare line to regular expression

```
    if  regular expression matches:
        print line
```

Setting up a regular expression

from the re module

compile the pattern

“Fred”

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

regular expression object

UCS

Skeleton Python script — 3

```
import re
pattern = "Fred"
regexp = re.compile(pattern)
for line in open('in.txt'):
    compare line to regular expression
    if  regular expression matches:
        print line
```

Prepare the regular expression

Using a regular expression

The reg. exp. object we just prepared.

The reg. exp. object's search() method.

The text being tested.

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

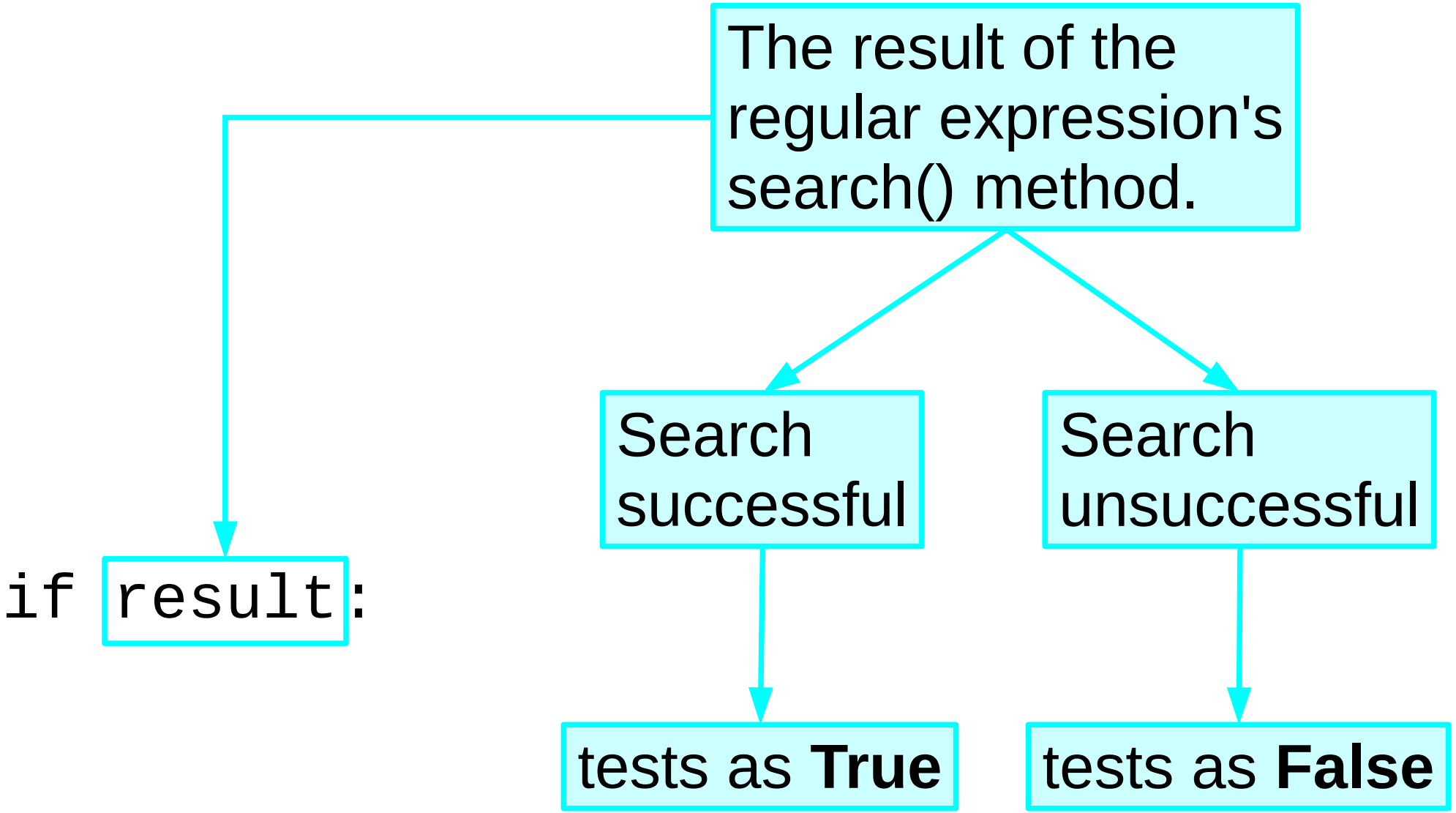
The result of the test.

Skeleton Python script — 4

```
import re
pattern = "Fred"
regexp = re.compile(pattern)
for line in open('in.txt'):
    result = regexp.search(line)
    if  regular expression matches :
        print line
```

Use the reg. exp.

Testing a regular expression's results



Skeleton Python script — 5

```
import re
pattern = "Fred"
regexp = re.compile(pattern)
for line in open('in.txt'):
    result = regexp.search(line)
    if result:
        print line
```

See if the line matched

Exercise : complete and test your file

```
$ python filter01.py
Fred
Freda
Frederick
```

Case sensitive matching

`names.txt`

Fred	✔
Freda	✔
Frederick	✔
Manfred	✘

Python matches are case sensitive by default

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

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.
…
```

`atoms.log`

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?

“Literal” text

Fixed text

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

Digits

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

Six digits

Letters

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

Sequence of lower case letters

And no more!

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

The line *starts* here

…and *ends* here

Building the pattern — 1

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

Start of the line marked with ^

An “anchored” pattern

```
^
```

Building the pattern — 2

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

Literal text

Don't forget the space!

```
^RUN␣
```

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

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}
```

Building the pattern — 5

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

Literal text
(with spaces).

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

Building the pattern — 6

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

Sequence of lower case letters

`[a-z]` "any single character between a and z"

`[a-z]+` "one or more characters between a and z"

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

Building the pattern — 7

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

Literal text

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

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.$
```

Exercise: running the filter

1. Copy `filter01.py` ⟶ `filter03.py`

2. Edit `filter03.py`

   Use the `^RUN...` regular expression.
   Use the atoms.log file

3. Test it

```
$ python filter03.py
```

Special codes in regular expressions

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

Examples of what can go in “[…]”

`[aeiou]` → any lowercase vowel

`[A-Z]` → any uppercase alphabetic

`[A-Za-z]` → any alphabetic

`[A-Za-z\]]` → any alphabetic or a ‘]’

backslashed character

`[A-Za-z\-]` → any alphabetic or a ‘-’

`[-A-Za-z]` → any alphabetic or a ‘-’

‘-’ as first character:
special behaviour for ‘-’ only

More...

`[^aeiou]` → *not* any lowercase vowel

`[^A-Z]` → *not* any uppercase alphabetic

`[\^A-Z]` → any uppercase alphabetic or a caret

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’.

What matches “[” ?

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

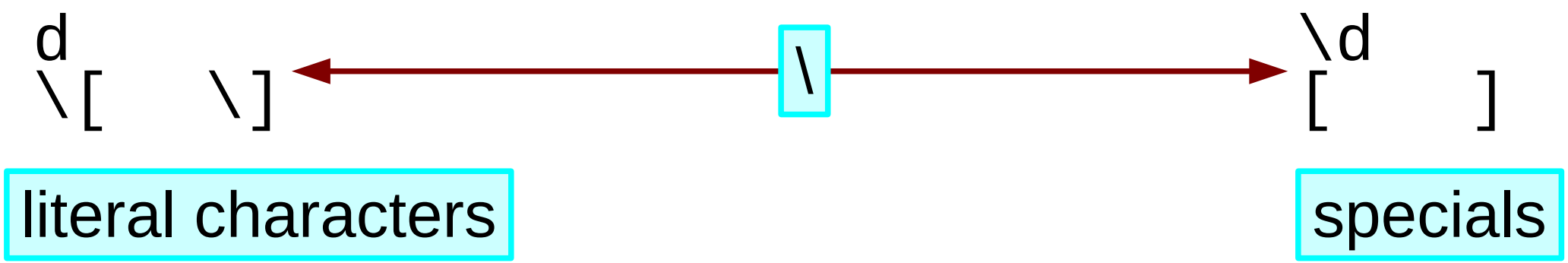
What matches “[abcd]”?

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

`\[abcd\]` → `[abcd]`

Backslash

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



What does dot match?

We've been using dot as a literal character.

Actually…

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

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

Building the pattern — 9

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

Actual full stops in the literal text.

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

Exercise

Input: `messages`

Script: `filter04.py`

Match lines with “Invalid user”.

```
Jun 25 23:47:33 noether sshd[9277]: Invalid user account from 207.54.140.124
Jun 25 23:47:34 noether sshd[9282]: Invalid user adam from 207.54.140.124
Jun 25 23:47:35 noether sshd[9287]: Invalid user adi from 207.54.140.124
Jun 25 23:47:36 noether sshd[9292]: Invalid user adina from 207.54.140.124
```

Match the *whole* line!
i.e., do a pattern for the information parts of the line.

UCS

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

The final string

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

```
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}$'
```

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

Problem/example

Our expression matches the Months

“Ann”, “Man”, “Dom”, “Aaa”, “Eea”, etc...

We want to fine tune it to

“Jan” or

“Feb” or

“Mar” or

“Apr” …

Alternation syntax

Must be in parentheses

A “group”

```
(Jan|Feb|Mar|Apr)
```

Separated by |

Parentheses in regular expressions

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

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

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

Hard to write

Harder to read

Hardest to maintain

Solutions

Multi-line layout

Comments

Verbose mode

Start raw long string

Backslashes!

Significant space

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

Comment

Ignored space

End long string

Telling Python to “go verbose”

Verbose mode

Another option, like ignoring case

Module constant, like `re.IGNORECASE`

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

```
import re
…
pattern = r"^[A-Z][a-z]{2}␣…$"
regexp = re.compile(pattern)
…
```

1 2 3

```
import re
…
pattern = r"""^
[A-Z][a-z]{2}\␣…
$"""
regexp = re.compile(pattern,re.VERBOSE)
…
```

Extracting parts 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.

Changing the pattern

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

Parentheses around the patterns

“Groups” again

The “match object”

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

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

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*

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.

Named groups

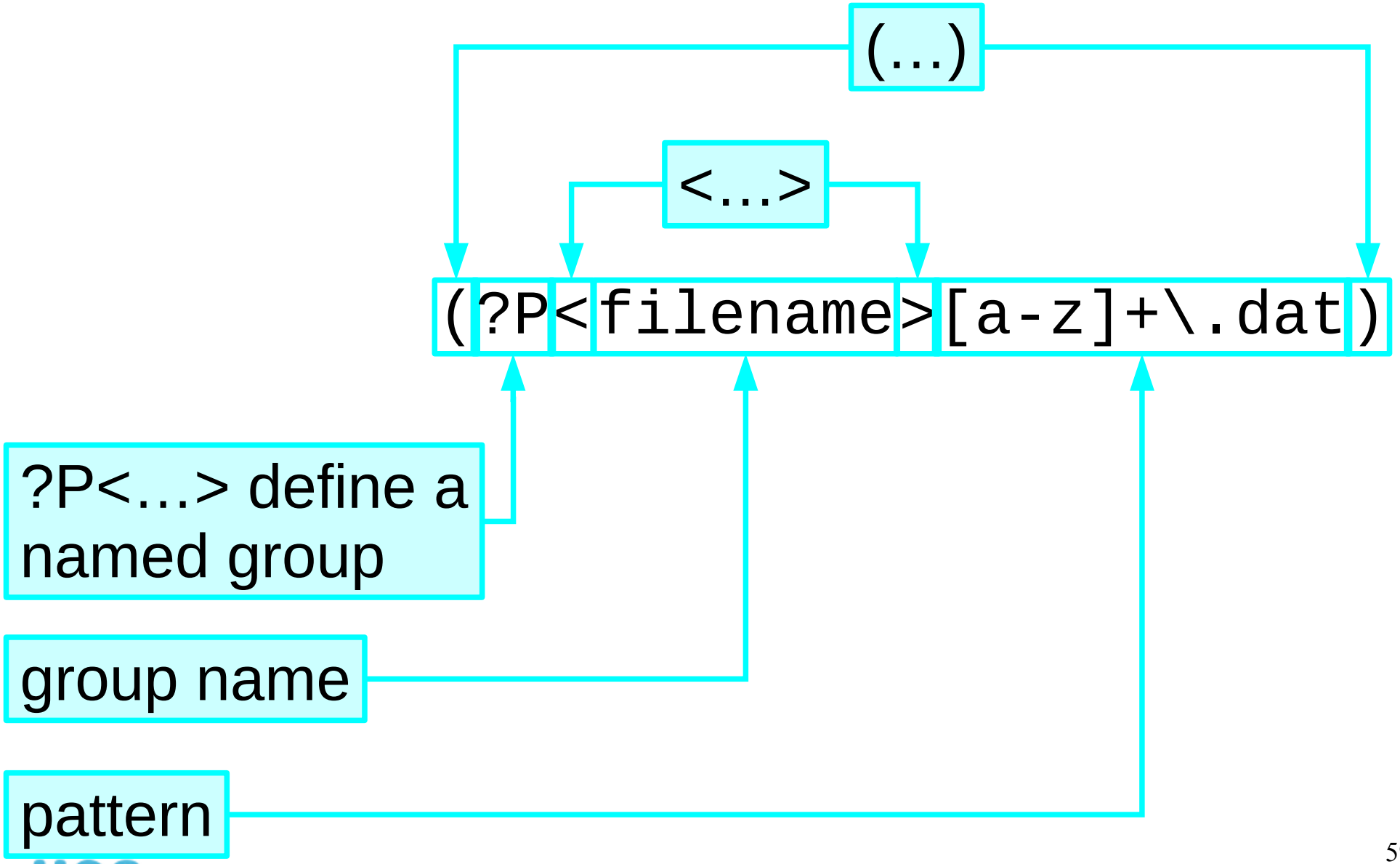
A group named "jobnum"

```
^                                   # Start of line
RUN\␣
(?P<jobnum>\d{6})                   # Job number
\␣COMPLETED\.\␣OUTPUT\␣IN\␣FILE\␣
(?P<filename>[a-z]+\.dat) # File name
\.
$                                   # End of line
```

Specifying the name

UCS

Naming a group



Using the named group

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

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

Parentheses in regular expressions

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

“Greedy” expressions

```
^([a-z]+)([a-z]+)$
```

```
$ python filter09.py
```

```
aa
aali
aardvar
aardvark
...
```

```
h
i
k
s
```

The first group is “greedy” at the expense of the second group.

Aim to avoid ambiguity

Hint to solve it: Use a “?” after the greedy operators e.g.: in ^([a-z]+?)([a-z]+)$ the first group is “lazy”

UCS

Multiple matches

Data: `boil.txt`

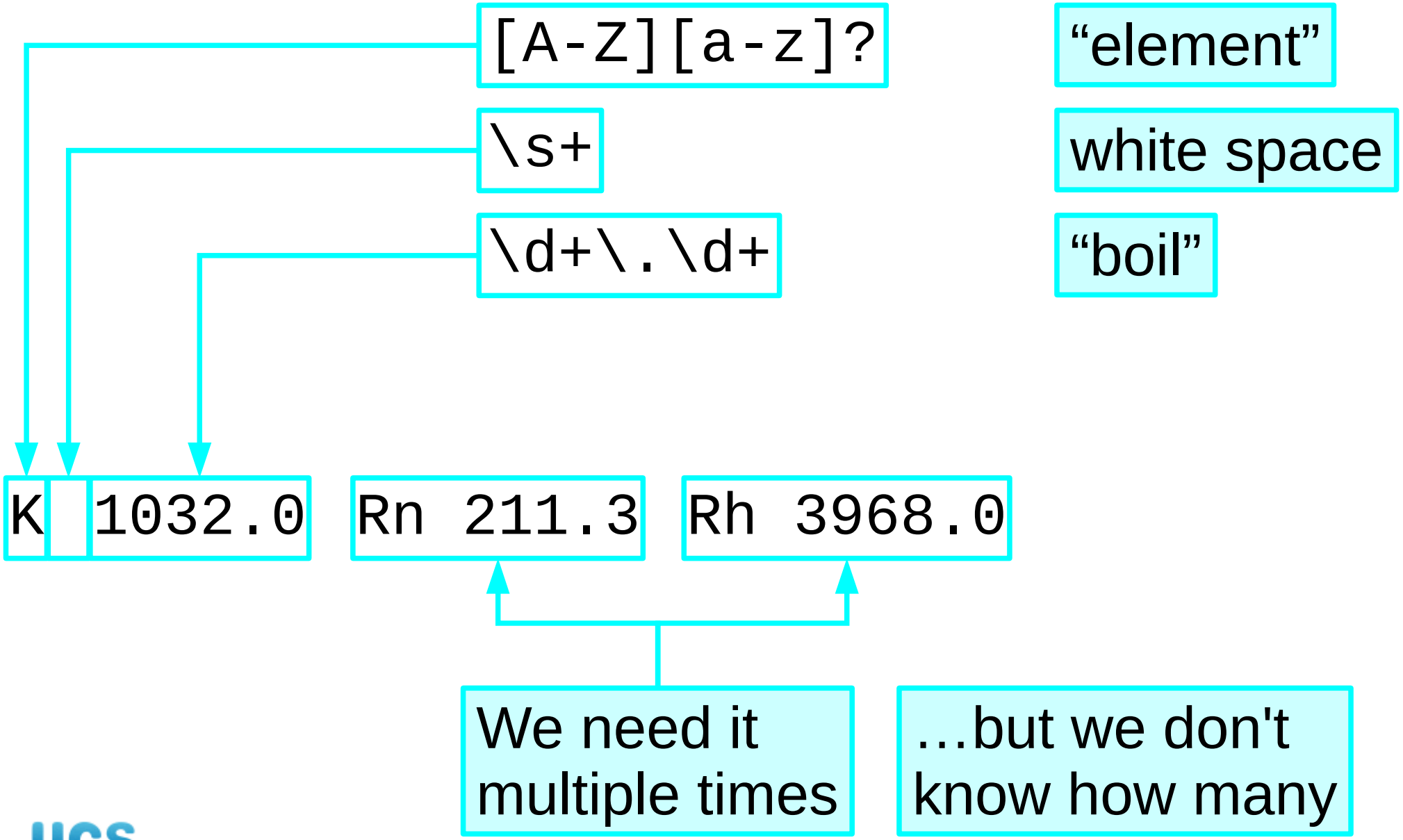
Basic entry: `Ar 87.3`

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

What pattern do we need?



Elementary pattern

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

Matches a single pair

Multiple matches

`regexp.search(line)` returns a *single* match

`regexp.finditer(line)` returns a *list* of matches

It would be better called searchiter() but never mind

Using finditer()

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

for line in ...:
    for match in regexp.finditer(line):
        …
```

Every matching case in each line