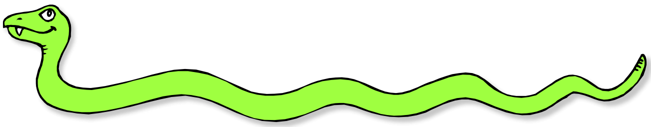
Quick Recap: Python

regular expressions

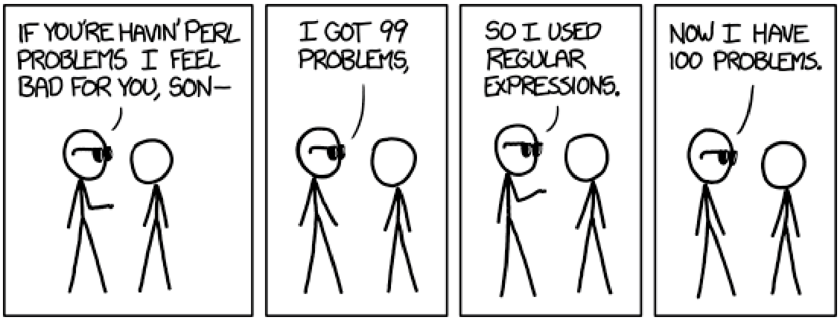


"Some people, when confronted with a problem, think

'I know, I'll use regular expressions.'

Now they have two problems."

-- Jamie Zawinski

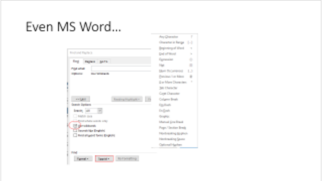

IF YOU'RE HAVIN' PERL PROBLEMS I FEEL BAD FOR YOU, SON—
I GOT 99 PROBLEMS,
SO I USED REGULAR EXPRESSIONS.
NOW I HAVE 100 PROBLEMS.

Motivation for Using Regular Expressions

- Text processing has been one of the most relevant topics since the dawn of computer science
- Regular Expressions are one of the most pervasive and useful tools/skills that you can possess for validation, search, extraction, and replacement of text
- Text ‘munging’ operations are greatly simplified by Regular Expressions (trust me…)

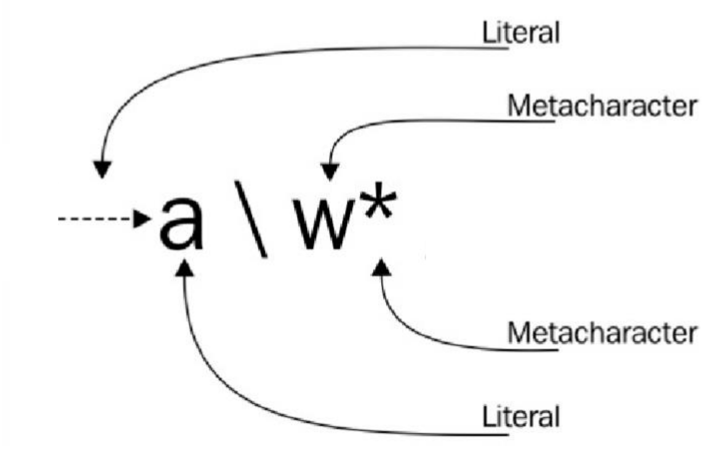
What are Regular Expressions?

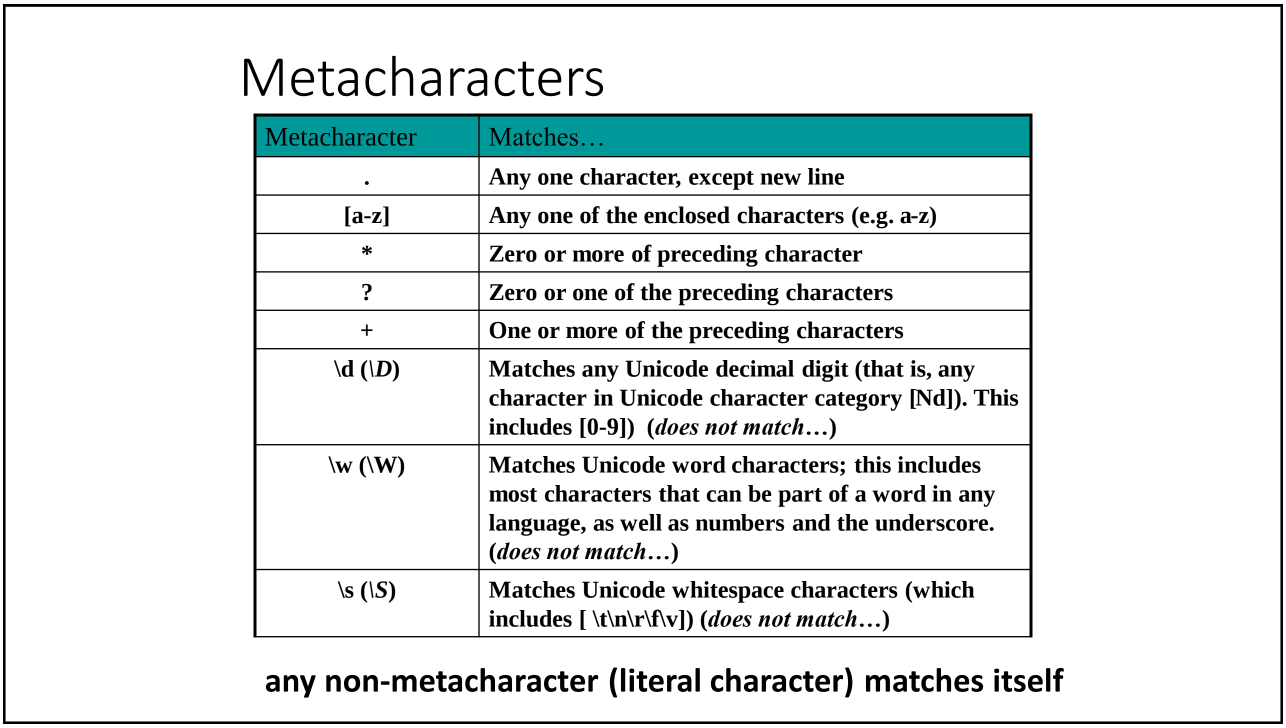
- A *pattern* of special characters used to match strings in a search
- Typically composed of a combination of literal characters and “special” characters called *metacharacters*
- Regular expressions are used, well, everywhere:
  - Editors: ed, ex, vi, emacs, Word (yes), NotePad++, Sublime, Visual Studio Code, Atom, PyCharm, etc…
  - Linux (WSL/bash) Utilities: grep, egrep, sed, and awk
  - Programming languages: python, perl, Java, Go, C, C#, C++, Ruby, PHP, Basic,…
- Use regular expressions to:
  - Search a string (search and match)
  - Replace parts of a string (sub)
  - Break strings into smaller pieces (split)



Even MS Word…

Regular Expression Syntax in a “Nutshell”

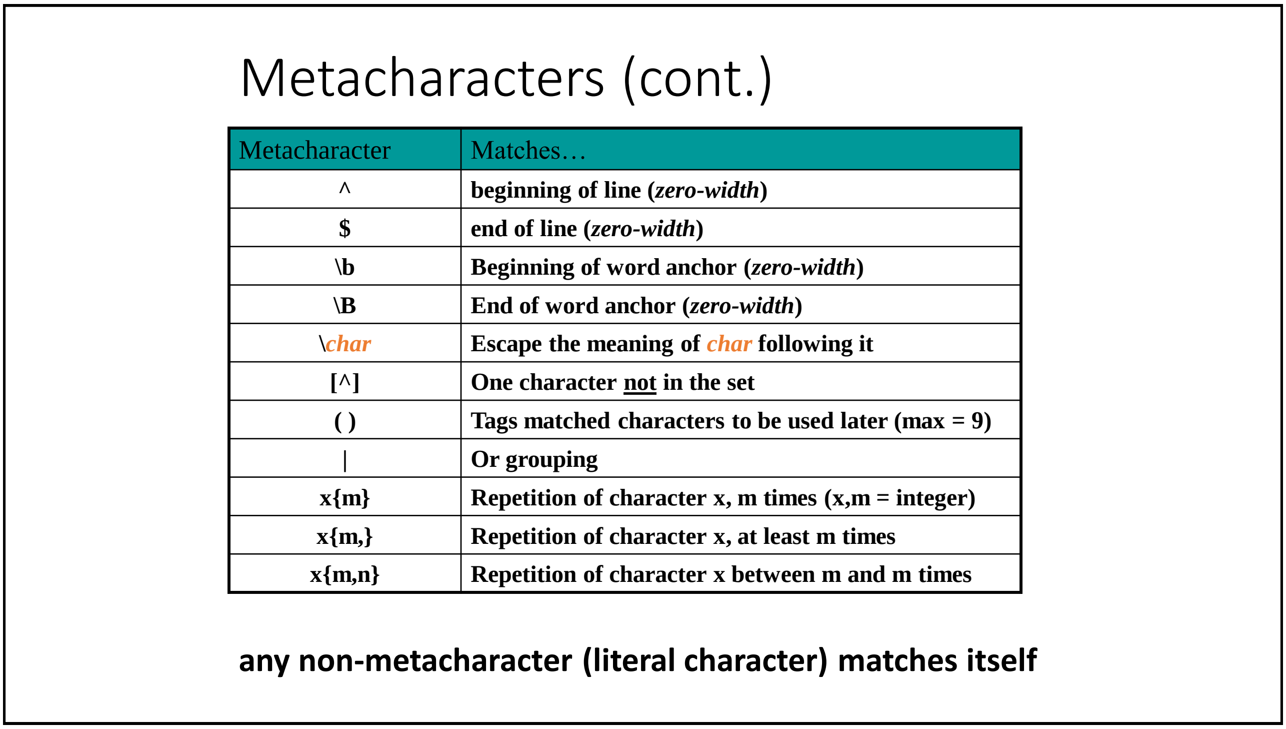




Metacharacters

Metacharacter	Matches…
.	**Any one character, except new line**
[a-z]	**Any one of the enclosed characters (e.g. a-z)**
*	**Zero or more of preceding character**
?	**Zero or one of the preceding characters**
+	**One or more of the preceding characters**
\d (\*D*)	**Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes [0-9]) (*does not match…*)**
\w (\W)	**Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. (*does not match…*)**
\s (\*S*)	**Matches Unicode whitespace characters (which includes [\t\n\r\f\v]) (*does not match…*)**

any non-metacharacter (literal character) matches itself

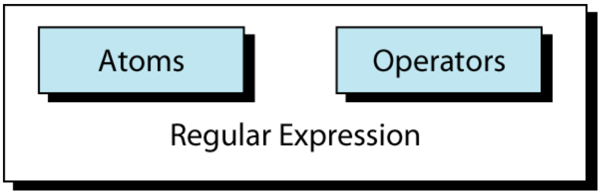


Metacharacters (cont.)

Metacharacter	Matches…
^	**beginning of line (*zero-width*)**
$	**end of line (*zero-width*)**
\b	**Beginning of word anchor (*zero-width*)**
\B	**End of word anchor (*zero-width*)**
\char	**Escape the meaning of *char* following it**
[^]	**One character not in the set**
( )	**Tags matched characters to be used later (max = 9)**
|	**Or grouping**
x{m}	**Repetition of character x, m times (x,m = integer)**
x{m,}	**Repetition of character x, at least m times**
x{m,n}	**Repetition of character x between m and m times**

any non-metacharacter (literal character) matches itself

Regular Expression

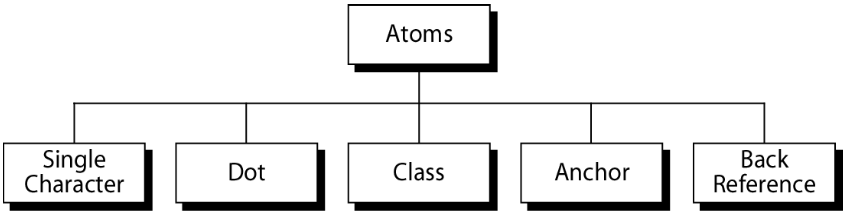


An atom specifies what text is to be matched and where it is to be found.

An operator **combines** regular expression atoms.

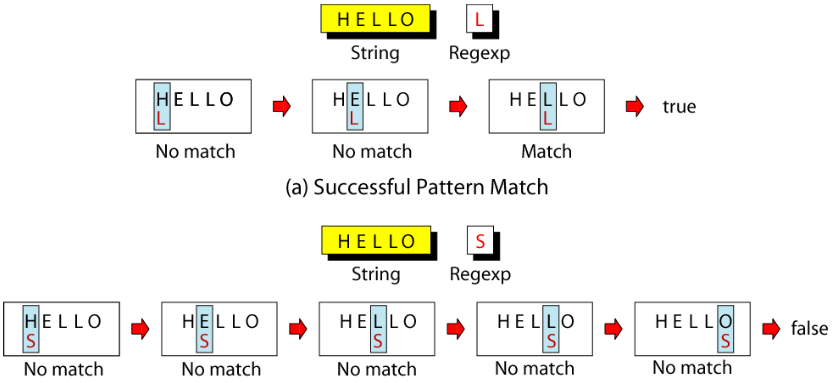
Atoms

An atom specifies what text is to be matched and where it is to be found.



Single-Character Atom

A single character matches itself

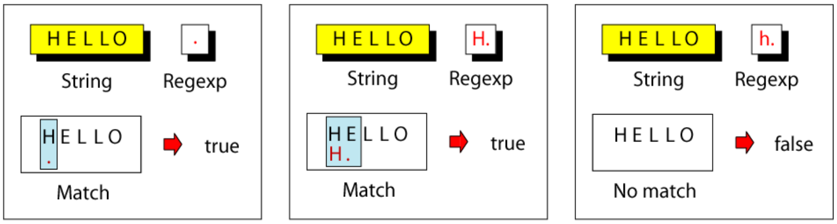


(a) Successful Pattern Match

(b) Unsuccessful Pattern Match

Dot Atom

matches any single character except for a new line character (\n)†



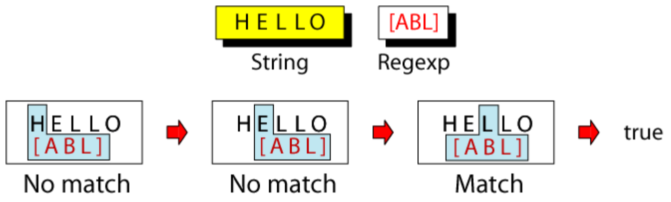
(a) Single-Character (b) Combination–True (c) Combination–False

† **Except when using re.DOTALL (re.S)**

Class Atom (aka – Character Class)

matches only single character that can be any of
the characters defined in a set:

Example: [ABC] matches either A, B, or C.



Notes:

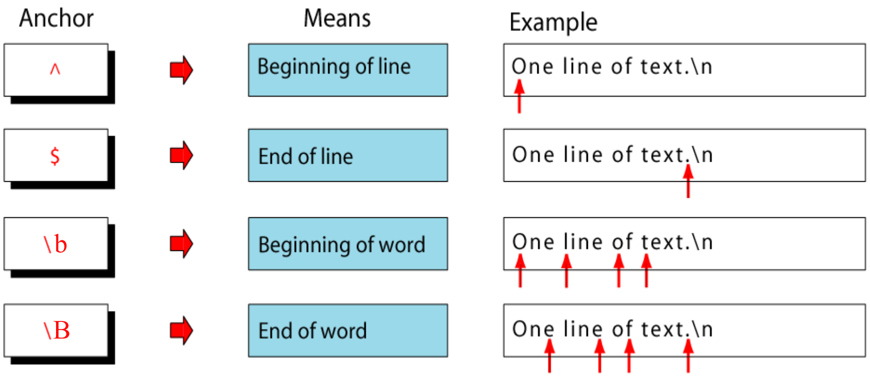
1) A range of characters is indicated by a dash, e.g. [A-Q]
2) Can specify characters to be excluded from the set, e.g. [^0-9] matches any character other than a number.

Example: Character Classes

RegExpr	Means	RegExpr	Means
[A-H]	[ABCDEFGH]	[^AB]	Any character except A or B
[A-Z]	Any uppercase alphabetic	[A-Za-z]	Any alphabetic
[0-9]	Any digit	[^0-9]	Any character except a digit
[[a]	[or a	[]a]	] or a
[0-9\-]	digit or hyphen	[^\^]	Anything except^

Anchor Atoms

Anchors tell where the next character in the pattern must be located in the text data. Anchors are ‘zero-width’.



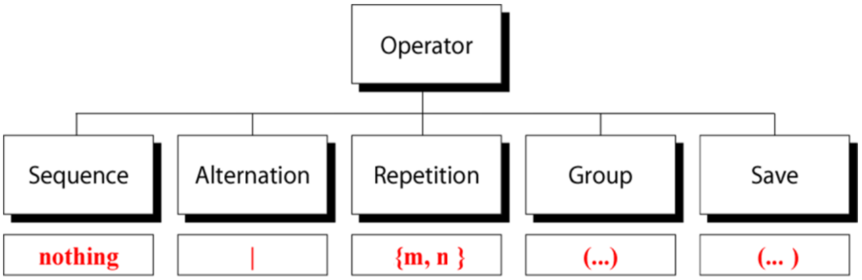
Anchor	Means	Example
^	Beginning of line	One line of text.\n
$	End of line	One line of text.\n
\b	Beginning of word	One line of text.\n
\B	End of word	One line of text.\n

Back Reference Atoms: \n

- used to retrieve saved text in one of 99 numbered “buffers”
- can refer to the text previously captured in a buffer by using a back reference:

  ex.: \1 \2 \3 ...\99

Operators



Operator

Sequence — nothing

Alternation — |

Repetition — {m, n }

Group — (...)

Save — (...)

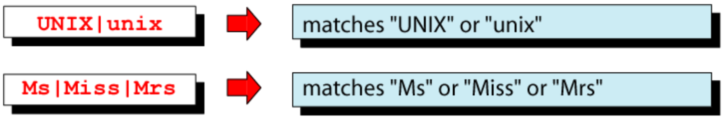
Sequence Operator

In a sequence operator, if a series of atoms are shown in a regular expression, there is no operator between them.

`dog`	➡	matches the pattern "dog"
`a..b`	➡	matches "a" , any two characters, and "b"
`[2-4][0-9]`	➡	matches a number between 20 and 49
`[0-9][0-9]`	➡	matches any two digits
`^$`	➡	matches a blank line
`^.$`	➡	matches a one-character line
`[0-9]-[0-9]`	➡	matches two digits separated by a "-"

Alternation Operator: |

operator (|) is used to define one
or more alternatives

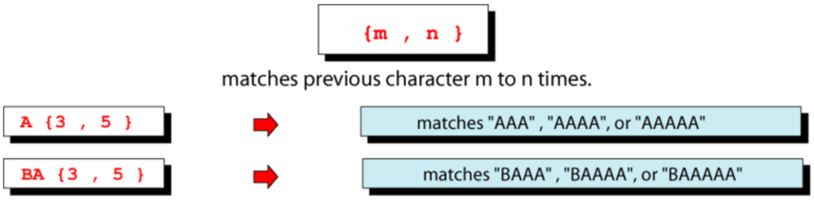


UNIX|unix → matches "UNIX" or "unix"

Ms|Miss|Mrs → matches "Ms" or "Miss" or "Mrs"

Repetition Operator: {…}
(aka – Quantifiers)

The repetition operator specifies that the atom or expression immediately before the repetition may be repeated.



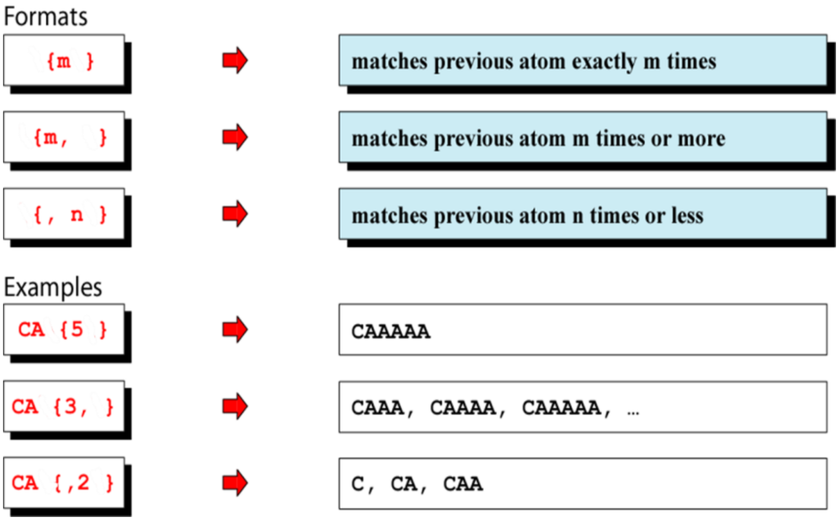
{m , n }

matches previous character m to n times.

A {3 , 5 } → matches "AAA" , "AAAA", or "AAAAA"

BA {3 , 5 } → matches "BAAA" , "BAAAA", or "BAAAAA"

Basic Repetition Forms



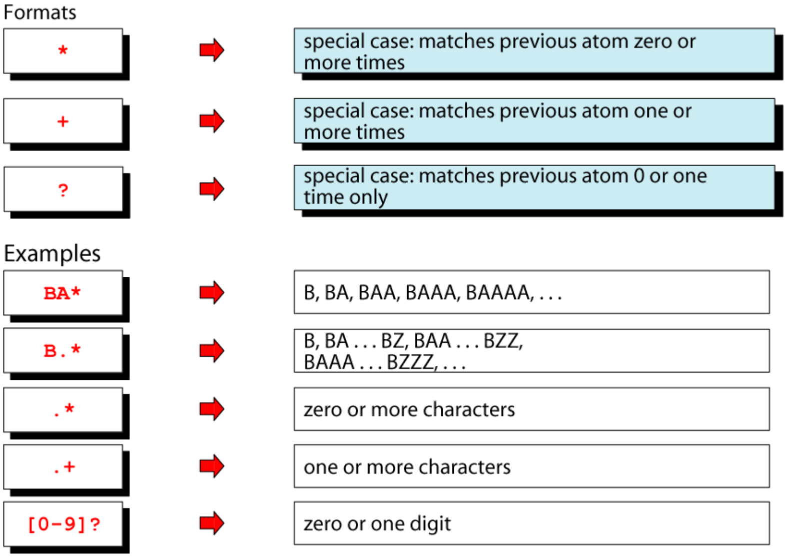
Formats

{m }	matches previous atom exactly m times
{m, }	matches previous atom m times or more
{, n }	matches previous atom n times or less

Examples

CA {5 }	CAAAAA
CA {3, }	CAAA, CAAAA, CAAAAA, …
CA {,2 }	C, CA, CAA

Short Form Repetition Operators: * + ?



Formats

*	special case: matches previous atom zero or more times
+	special case: matches previous atom one or more times
?	special case: matches previous atom 0 or one time only

Examples

BA*	B, BA, BAA, BAAA, BAAAA, . . .
B.*	B, BA . . . BZ, BAA . . . BZZ, BAAA . . . BZZZ, . . .
.*	zero or more characters
.+	one or more characters
[0-9]?	zero or one digit

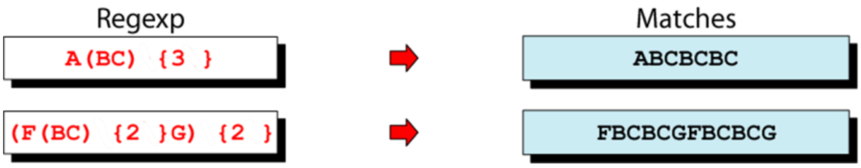
Greedy and Reluctant (non-greedy) Quantifiers

Pattern	String	Match
r'.*'	Hello	Hello
r'.*?'	Hello	H

- By default, quantifiers are "greedy"
- A greedy quantifier will try to match as much as possible to have the largest match result possible
- The reluctant (non-greedy) quantifier will behave the exact opposite
- A reluctant quantifier will try to have the smallest match possible
- Non-greedy behavior can be requested by adding an extra question mark to the quantifier; for example, ??, *? or +?.

Group Operator

In the group operator – (…), when a group of characters is enclosed in parentheses, the next operator applies to the whole group, not only the previous characters.



Regexp

Matches

A(BC)\{3\} → ABCBCBC

(F(BC)\{2\}G)\{2\} → FBCBCGFBCBCG

The group operator also "captures" its matches in sequentially numbered groups (buffers) starting at 1. Captured matches can be referenced in the RE using Back References such as \1, \2, etc.

More on “Capturing” Groups…

```
(?P<name>...)
```

- Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name *name*. Group names must be valid Python identifiers.

```
(?P=name>)
```

- A back reference to a named group; it matches whatever text was matched by the earlier group named *name*. Named groups still retain their back reference numbers in addition to their names

Consider the pattern:

```
(?P<quote>['"]).*?(?P=quote)
```

-OR-

```
 (['"]).*?(\1)
```

What’s going on here? What’s that question mark doing again?

Python’s re.Search and re.Match

- The two basic re matching functions are *re.search* and *re.match*
  - Search looks for a pattern anywhere in a string
  - Match looks for a match starting at the beginning of a string
- Both return *None* (BTW, None evals to False in a conditional test) if the pattern isn’t found and a **match object** instance if it is found

```
>>> import re
>>> pat = "a*b”
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
>>> <None>
```

What’s a match object?

- an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group() returns string matched
'aaab’
>>> r1.group(0)  # group(0) same thing
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

- Here’s a pattern to match simple email addresses
  \w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"robertsj1503@duq.edu")
>>> r1.group(0)
'robertsj1503@duq.edu’
```

- We might want to extract the pattern parts, like the email name and host?

What got matched?

- We can put parentheses around groups we want to be able to reference

```
>>> pat2 = r"(\w+)@((\w+\.)+(com|org|net|edu))"
>>> r2 = re.match(pat2,"robertsj1503@duq.edu")
>>> r2.group(1)
‘robertsj1503'
>>> r2.group(2)
‘duq.edu’
```

Note: ‘groups’ are numbered in a preorder traversal of the forest – outside-in, then left to right (it is possible to have groups within groups…)

```
>>> r2.groups()
('robertsj1503', 'duq.edu', 'duq.', 'edu’)
```

Note: does not include group(0) (the entire match)

What got matched?

- We can ‘label’ the groups as well…

```
>>> pat3 =r"(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|net|edu))"
>>> r3 = re.match(pat3,”robertsj1503@duq.edu”)
>>> r3.group('name')
‘robertsj1503'
>>> r3.group('host')
‘duq.edu’
```

- And reference the matching parts by the labels

More re functions

- re.split() is like str().split but can use patterns

```
>>> re.split("\W+", “This... is a test,short and sweet, of split().”)
['This', 'is', 'a', 'test', 'short’,
  'and', 'sweet', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue socks and red shoes')
'black socks and black shoes’
```

- re.findall() finds all matches returning an iterable

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("robertsj1503@duq.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'robertsj1503@duq.edu'
```

Pattern object methods

Pattern objects have methods that parallel the re functions (e.g., match, search, split, findall, sub), e.g.:

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
>>> p2 = re.compile("[.?!]+\s+")
>>> p2.split("Tired? Go to bed!   Now!! ")
['Tired', 'Go to bed', 'Now']
```

email address

sentence boundary

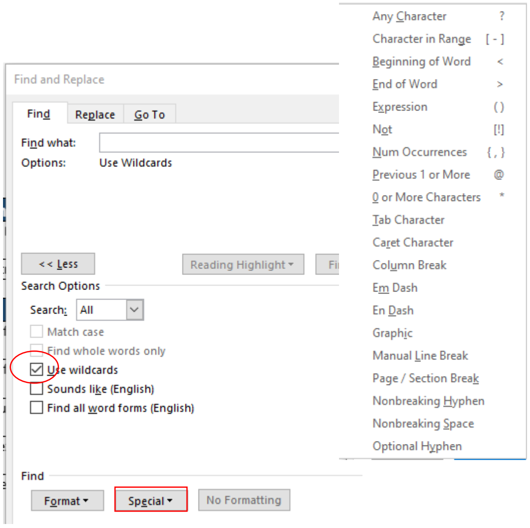
Where to go from here?

- Mastering Regular Expressions, by Jeffrey Friedl. Link
  - For those aiming for ‘RegEx Guru’ status
- Online docs
- Experiment!
  - https://regex101.com/
  - This may be the most useful tool in your RE arsenal!

Try It…

- Complete the regular expression lab in the Canvas module *Unit - Manipulating Text Sequence Data*

Even MS Word…



Find what:

Options: Use Wildcards

Search: All

□ Match case

□ Find whole words only

☑ Use wildcards

□ Sounds like (English)

□ Find all word forms (English)