Lecture 24: Regular Expressions

Aug 2nd, 2021

Alex Kassil

Announcements

- Guest Lecture **Tuesday, August 3rd, 6pm-7pm Pacific Time.**
  - The guest lecturer is Igor Podkhodov, Senior Engineering Manager at Zoox! He has been working in the field of technology for nearly 20 years now, and has been a Software Engineer/Manager at companies like Samsung, Netflix, Google, Snapchat, and Facebook before Zoox.
  - Submit questions in advance here: https://links.cs61a.org/guest-lecture-questions
- Vitamin 10 due tomorrow 8am
- Vitamin 11 due tomorrow 8am
- Lab 11 due tomorrow 11:59pm
- HW 06 due Wednesday 11:59pm
- Scheme Project
  - submit with Questions 1-6 done by Tuesday, 8/3 (worth 1 pt), and
  - submit with Parts I and II complete (including passing all additional tests provided in `tests.scm`) by Friday, 8/6 (worth 1 pt), and
  - submit the entire project by Tuesday, 8/10. You will get an extra credit point for submitting the entire project by Monday, 8/9

Regular Expressions (Regex) Basics

Can be pronounced rejex or reggex. I think reggex is more right,
but rejex is easier for me to say and sounds better to me.

Pattern Matching

- Programs that manipulate text often have a need to search a string for things other than simple substrings.
- For example: “Find all numbers in this string” or “Find all Scheme tokens in this program text.”
- Another application might be to check input: “Does this user’s response have the proper form?” or “Did this user enter enough digits for their phone number?”
- We can think of this as a kind of declarative programming, because the programmer is saying, e.g., “find something that looks like this” rather than “search for the substring ‘(’, then look for a ’)’ after that” to check for a parenthesized expression.

What are Regular Expressions?

- One of the most widely available and useful mechanisms is the *regular expression*.
- Formally, regular expressions denote *sets of strings* that are called *regular languages*.
- But normally, we think of them as patterns that match certain strings.
- In Python, we denote them with strings and use them as patterns by means of functions and classes in the module `re`.
- We will spend some time building up to the example below that extracts information from a date string

```
>>> import re
>>> date = "January 1st, 1970 00:00:00"
>>> re.match(r"(\w+) (\d+\w+), (\d+) (\d+):(\d+):(\d+)", date).groups()
('January', '1st', '1970', '00', '00', '00')
```

Regular Expression Syntax

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string

Regular Expression Syntax

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string

Regular Expression Syntax

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA

Regular Expression Syntax

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Regular Expression Syntax

AB*: A then zero or more copies of B: A, AB, ABB, ABBB
(AB)*: Zero or more copies of AB: ABABABAB, ABAB, AB,

Also matches the empty string!

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

regex101.com favorite tool for working with regex

Let’s write a regular expression that matches scheme with an odd number of e’s between sch and me.

Valid:

scheme, scheeeme, scheeeeeme, …

Not valid:

schme, scheeme, python, ...

regex101.com favorite tool for working with regex

Let's write a regular expression that matches scheme with an odd number of e's between sch and me.

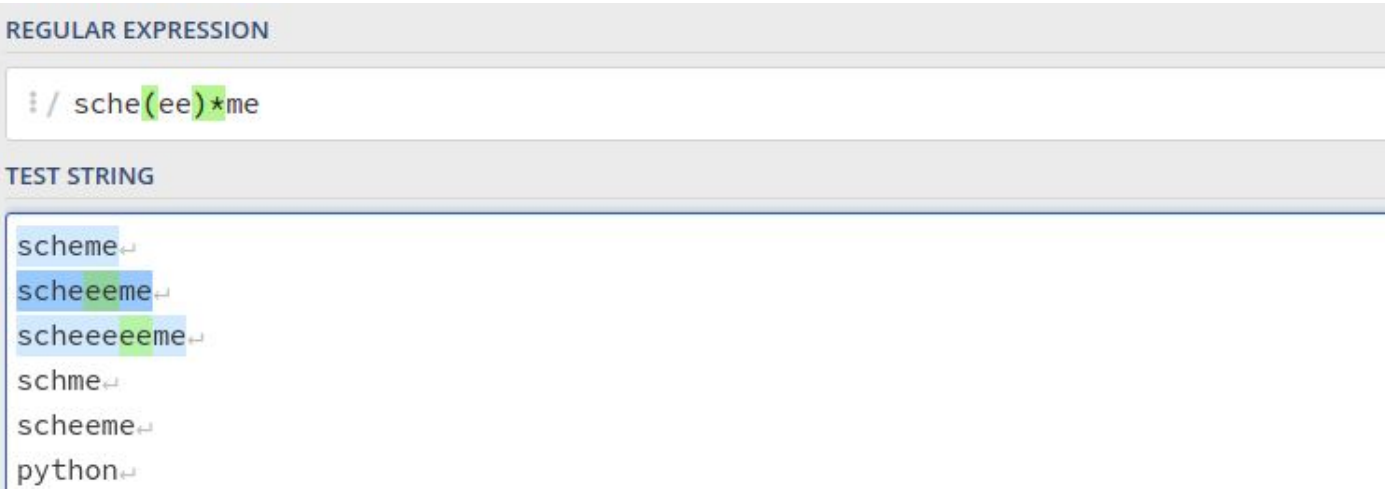
Valid:

scheme, scheeeme, scheeeeeme, …

Not valid:

schme, scheeme, python, … solution:

https://r



https://regex101.com/r/ApznBC/1 Your turn!

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Write a regular expression that matches scheme with an odd number of e's between sch and me or an even number of ea's except zero.

Valid: scheme, scheeeme, scheame, scheaeame

Invalid: scheeme, schme, python, schame, schaeme

https://regex101.com/r/ekS1AC/1 Solution!

`sch(e(ee)*|ea(ea)*)me`

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Write a regular expression that matches scheme with an odd number of e's between sch and me or an even number of ea's except zero.

Valid: scheme, scheeeme, scheame, scheaeame

Invalid: scheeme, schme, python, schame, schaeme

Order of Operations in Regexes

`sch(e(ee)*|ea(ea)*)me`

- Matches starting with sch and ending with me, with either of the following in the middle:
  - `e(ee)*`
  - `ea(ea)*`

Match examples:

`scheme`
`scheeeme`
`scheame`
`scheaeame`

Order of Operations in Regexes

`sch(e(ee)*|ea(ea)*)me`

- Matches starting with sch and ending with me, with either of the following in the middle:
  - `e(ee)*`
  - `ea(ea)*`

Match examples:

```
scheme
scheeeme
scheame
scheaeame
```

`sch(e(ee)*)|(ea(ea)*)me`

- Matches either of the following
  - `sch` followed by `e(ee)*`
  - `ea(ea)*` followed by `me`

Match examples:

```
sche
sche
eame
eaeame
```

https://regex101.com/r/oyjGnW/1

In regexes | comes last.

https://regex101.com/r/ApznBC/1 Your turn!

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Write a regular expression that matches scheme with an odd number of e's between sch and me or an even number of ea's except zero.

Valid: scheme, scheeeme, scheame, scheaeame

Invalid: scheeme, schme, python, schame, schaeme

Matching some CS Lower Divs

Write a regular expression that matches strings that start with CS61 and end with a capital letter

Valid: CS61A, CS61B, CS61C, CS61D, ..., CS61Z

Not valid: CS70, CS10, CS611, EECS16A, ...

```
(CS61A|CS61B|CS61C|CS61D|CS61E|CS61F|CS61G|CS61H|CS61I|CS61J|CS61K|CS61L|CS61M|CS61N|CS61O|CS61P|CS61Q|CS61R|CS61S|CS61T|CS61U|CS61V|CS61W|CS61X|CS61Y|CS61Z)
```

```
CS61(A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)
```

Writing such long expressions is a little tedious

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
inverse character class	`[^A-Za-z]*`	`12345` `?!*%3`	`word` `BANANAS`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
inverse character class	`[^A-Za-z]*`	`12345` `?!*%3`	`word` `BANANAS`
at least one	`jo+hn`	`john` `jooooooohn`	`jhn` `jjohn`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
inverse character class	`[^A-Za-z]*`	`12345` `?!*%3`	`word` `BANANAS`
at least one	`jo+hn`	`john` `jooooooohn`	`jhn` `jjohn`
zero or one	`joh?n`	`jon` `john`	`any other string`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
inverse character class	`[^A-Za-z]*`	`12345` `?!*%3`	`word` `BANANAS`
at least one	`jo+hn`	`john` `jooooooohn`	`jhn` `jjohn`
zero or one	`joh?n`	`jon` `john`	`any other string`
repeated exactly {a} times	`j[aeiou]{3}hn`	`jaoehn` `jooohn`	`jhn` `jaeiouhn`

Expanded Regex Syntax

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	`BANANAS` `AAAAAAA`	`ALABAMA` `AAAAAA`
character class	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
inverse character class	`[^A-Za-z]*`	`12345` `?!*%3`	`word` `BANANAS`
at least one	`jo+hn`	`john` `jooooooohn`	`jhn` `jjohn`
zero or one	`joh?n`	`jon` `john`	`any other string`
repeated exactly {a} times	`j[aeiou]{3}hn`	`jaoehn` `jooohn`	`jhn` `jaeiouhn`
repeated from a to b times: {a,b}	`j[ou]{1,2}hn`	`john` `juohn`	`jhn` `jooohn`

Your turn! https://regex101.com/r/07v7K1/1

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	BANANAS AAAAAAA	ALABAMA AAAAAA
character class	`[A-Za-z][a-z]*`	word Capitalized	camelCase 4illegal
inverse character class	`[^A-Za-z]*`	12345 ?!*%3	word BANANAS
at least one	`jo+hn`	john jooooooohn	jhn jjohn
zero or one	`joh?n`	jon john	any other string
repeated exactly {a} times	`j[aeiou]{3}hn`	jaoehn jooohn	jhn jaeiouhn
repeated from a to b times: {a,b} {a,} means a or more	`j[ou]{1,2}hn`	john juohn	jhn jooohn

Match social security numbers, example 111-11-1111

Your turn! https://regex101.com/r/07v7K1/1

operation	example	matches	does not match
any character (except newline)	`.A.A.A.`	BANANAS AAAAAAA	ALABAMA AAAAAA
character class	`[A-Za-z][a-z]*`	word Capitalized	camelCase 4illegal
inverse character class	`[^A-Za-z]*`	12345 ?!*%3	word BANANAS
at least one	`jo+hn`	john jooooooohn	jhn jjohn
zero or one	`joh?n`	jon john	any other string
repeated exactly {a} times	`j[aeiou]{3}hn`	jaoehn jooohn	jhn jaeiouhn
repeated from a to b times: {a,b} {a,} means a or more	`j[ou]{1,2}hn`	john juohn	jhn jooohn

Match social security numbers, example 111-11-1111
Solution: `[0-9]{3}-[0-9]{2}-[0-9]{4}`

Simple Email Address Regex

Let’s write a regular expression that matches email addresses of the form:

letters@letters.exactly 3 letters

Valid: alex@gmail.com, gmail@alex.com, alexkassil@berkeley.edu

Not valid: alex@gmail, gmail+org@gmail.com, ALEXKASSIL@BERKELEY.EDU

Simple Email Address Regex

Let's write a regular expression that matches email addresses of the form:

letters@letters.exactly 3 letters

Valid: alex@gmail.com, gmail@alex.com, alexkassil@berkeley.edu

Not valid: alex@gmail, gmail+org@gmail.com, ALEXKASSIL@BERKELEY.EDU

Solution: `[a-z]+@[a-z]+\.[a-z]{3}`

Email Address Regular Expression (a probably bad idea)

The regular expression for email addresses (for the Perl programming language):

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?: \r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\ ](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n) ?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t] )*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])* )(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*) *:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r \n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t ]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]( ?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(? :\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)? [ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]| \\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|" (?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\ ".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[ \]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|( ?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([ ^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\ ]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\] |\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\ .|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(? :[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\". \[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\] ]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[ \["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*))*)?;\s*)
```

From: http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Built in Character Classes

pattern	explanation	matches
\d	`Any single digit, [0-9]`	1 2

Built in Character Classes

pattern	explanation	matches
\d	Any single digit, [0-9]	1 2
\w	Any single letter, digit or underscore, [A-Za-z0-9_]	A _

Built in Character Classes

pattern	explanation	matches
\d	Any single digit, [0-9]	1 2
\w	Any single letter, digit or underscore, [A-Za-z0-9_]	A _
\W	Anything \w doesn't match, [^A-Za-z0-9_]	! φ

Built in Character Classes

pattern	explanation	matches
\d	Any single digit, [0-9]	1 2
\w	Any single letter, digit or underscore, [A-Za-z0-9_]	A _
\W	Anything \w doesn't match, [^A-Za-z0-9_]	! φ
\s	Any single whitespace character: space, tab, newline, carriage return, "\f", or "\v"	

Built in Character Classes

pattern	explanation	matches
\d	Any single digit, [0-9]	1 2
\w	Any single letter, digit or underscore, [A-Za-z0-9_]	A _
\W	Anything \w doesn't match, [^A-Za-z0-9_]	! φ
\s	Any single whitespace character: space, tab, newline, carriage return, "\f", or "\v"	
\S	Any single character that is not whitespace	A φ

Regex Makeover!

Let's make a regular expression to match 24-hour times of the format `HH:MM`.

First draft: `[0-2]\d:\d\d`

What not valid times would that match?

`25:99`

How do we fix minutes?

`[0-2]\d:[0-5]\d`

How do we fix hours?

`(2[0-3]|[0-1]\d):[0-5]\d`

Anchors

A few patterns match the empty string, but only at certain places.

pattern	explanation	example	matches	does not match
`^`	`Matches the empty string at the beginning of a string.`	`^hello`	**hello** **hello** there	`why hello`

Anchors

A few patterns match the empty string, but only at certain places.

pattern	explanation	example	matches	does not match
`^`	Matches the empty string at the beginning of a string.	`^hello`	**hello** **hello** there	why hello
`$`	Matches the empty string at the end of a string.	`hello$`	**hello** why **hello**	hello there

Anchors

A few patterns match the empty string, but only at certain places.

pattern	explanation	example	matches	does not match
^	Matches the empty string at the beginning of a string.	^hello	**hello** **hello** there	why hello
$	Matches the empty string at the end of a string.	hello$	**hello** why **hello**	hello there
\b	Matches the empty string at the beginning or end of a word (composed of matches to \w).	\b4\b	44 pieces of a4 is **4**$	44 pieces of a4

Anchors

A few patterns match the empty string, but only at certain places.

pattern	explanation	example	matches	does not match
^	Matches the empty string at the beginning of a string.	^hello	**hello** **hello** there	why hello
$	Matches the empty string at the end of a string.	hello$	**hello** why **hello**	hello there
\b	Matches the empty string at the beginning or end of a word (composed of matches to \w).	\b4\b	44 pieces of a4 is **4**$	44 pieces of a4
\B	Matches the empty string where \b does not match.	\B4\b	4**4** pieces of a**4** is 4 dollars	4a 4

Your Turn! https://regex101.com/r/3x1lsx/1

pattern	explanation	example	matches	does not match
^	Matches the empty string at the beginning of a string.	^hello	**hello** **hello** there	why hello
$	Matches the empty string at the end of a string.	hello$	**hello** why **hello**	hello there
\b	Matches the empty string at the beginning or end of a word (composed of matches to \w).	\b4\b	44 pieces of a4 is **4**$	44 pieces of a4
\B	Matches the empty string where \b does not match.	\B4\b	4**4** pieces of a**4** is 4 dollars	4a 4

Make a regular expression that matches all the lowercase words ending with ing

Your Turn! https://regex101.com/r/D6MkkM/1

pattern	explanation	example	matches	does not match
^	Matches the empty string at the beginning of a string.	^hello	**hello** **hello** there	why hello
$	Matches the empty string at the end of a string.	hello$	**hello** why **hello**	hello there
\b	Matches the empty string at the beginning or end of a word (composed of matches to \w).	\b4\b	44 pieces of a4 is **4**$	44 pieces of a4
\B	Matches the empty string where \b does not match.	\B4\b	4**4** pieces of a**4** is 4 dollars	4a 4

Make a regular expression that matches all the lowercase words ending with ing

Solution: `\b[a-z]+ing\b`

Escaping Characters

- Recap: Patterns that don't contain any of the special characters

  ```
  \ ( ) [ ] { } + * ? | $ ^ .
  ```

  simply match themselves

- Example: `Berkeley, CA 94720` matches exactly the string or substring `Berkeley, CA 94720`
- To match one of the special characters above, precede with a backslash
- Example: `\(1\+3\)` matches exactly `(1+3)`

Regular Expressions in Python

Small Preliminary: Raw Strings

- Traditionally, the backslash character (\) is often used in patterns.
- This can conflict with the usual Python string escape sequences (which begin with backslashes), like \n for newline
- So early on, Python introduced raw strings, which have an ‘r’ or ‘R’ in front of the quotes, as in `r"\n"`.
- In these strings, backslashes are just backslashes (except, annoyingly, that they cannot appear alone at the end of a string.)
- So generally, we use raw strings to denote patterns in Python that have backslashes.
- Reminder, strings in python prefixed by ‘f’ are formatted strings, which allow variables enclosed in { } to be evaluated and inserted into the string

Raw String Examples

```
>>> "\n"
'\n'
>>> r"\n"
'\\n'
>>> print("I have\na newline in me.")
I have
a newline in me
>>> print(r"I have\na newline in me.")
I have\na newline in me.
```

Using Patterns in Python

- Need to import the re module `import re`
- The methods `re.match`, `re.search`, and `re.fullmatch` all take a string containing a regular expression and a string of text. They return either a *match object* or, if there is no match, `None`.
- Match objects are 'true' values as far as Python is concerned, so one can use the results of these functions as True/False values:

```
>>> import re
>>> for x in ["jack", "25", "-5", "aardvark"]:
...   if re.fullmatch(r'-?\d+', x):
...     print(f"{x} is a number")
25 is a number
-5 is a number
>>> bool(re.fullmatch(r'-?\d+', '123'))
True
>>> bool(re.fullmatch(r'-?\d+', '123 people'))
False
```

The Matching Methods

- `re.fullmatch` requires that the pattern match the entire searched string
- `re.match` does not require that the whole string be matched, but does require that the matching string occur at the beginning of the string
- `re.search` finds the first occurrence of the pattern anywhere in the string

```
>>> import re
>>> x = "Structure and Interpretation of Computer Programs"
>>> bool(re.match("Structure", x))
True
>>> bool(re.fullmatch("Structure", x))
False
>>> bool(re.fullmatch("Structure.*Programs", x))
True
>>> bool(re.match("and", x))
False
>>> bool(re.search("and", x))
True
```

Write a function that checks if input string is a float

```
import re
def is_float(x):
    """ Return whether a string x is a float
    >>> is_float("0.0")
    True
    >>> is_float("0")
    False
    >>> is_float("-1234.5678")
    True
    >>> is_float("Chapter 2.1")
    False
    >>> is_float("1.1.1")
    False
    """
    pattern = r"_______"
    return bool(re._______(pattern, x))
```

Write a function that checks if input string is a float

```
import re
def is_float(x):
    """ Return whether a string x is a float
    >>> is_float("0.0")
    True
    >>> is_float("0")
    False
    >>> is_float("-1234.5678")
    True
    >>> is_float("Chapter 2.1")
    False
    >>> is_float("1.1.1")
    False
    """
    pattern = r"-?\d+\.\d+"
    return bool(re._______(pattern, x))
```

Write a function that checks if input string is a float

```
import re
def is_float(x):
    """ Return whether a string x is a float
    >>> is_float("0.0")
    True
    >>> is_float("0")
    False
    >>> is_float("-1234.5678")
    True
    >>> is_float("Chapter 2.1")
    False
    >>> is_float("1.1.1")
    False
    """
    pattern = r"-?\d+\.\d+"
    return bool(re.fullmatch(pattern, x))
```

Retrieving Matched Text

- Match objects also carry information about what has been matched. The `.group()` method allows you to retrieve it
- Furthermore, if there are parenthesized expressions in the pattern, you can retrieve them as well by indexing `.group()` or calling `.groups()`

```
>>> x = "This string contains 35 characters."
>>> mat = re.search(r'\d+', x)
>>> mat.group()
'35'
>>> x = "There were 12 pence in a shilling and 20 shillings in a pound."
>>> mat = re.search(r'(\d+).*(\d+)', x)
>>> mat.group(0) # Same as mat.group()
'12 pence in a shilling and 20'
>>> mat.group(1)
'12'
>>> mat.group(2)
'20'
>>> mat.groups() # All parenthesized groups
('12', '20')
```

For more information

- Sp21 Intro to Regex Slides
- Sp21 Review: Regular Expressions + BNF (Ignore the BNF part)
- https://regexone.com/ Online tutorial
- https://regex101.com/ platform for experimenting with regular expressions
- https://regexr.com/ online tool to learn, build, & test Regular Expressions
- https://regexcrossword.com/ Fun games to learn regex
- http://www.regular-expressions.info/
- https://projects.lukehaas.me/regexhub/
- Fa20 Data 100 Regex Reference
- Data 100 Textbook Section on Regex
- Sp21 Data 100 Regular Expressions Lecture