

# Lecture 22: Regular Expressions

July 27th, 2022

Cooper Bedin

# Announcements

- Ants due tonight!! You can still turn in by Saturday for 75% credit
- Magic: The Lambda-ing also due tonight! This is an EC project so is not eligible for extensions for most students
- HW 5 and Lab 9 due tonight
- We ran into a small technical issue with the CATS autograder—please make sure to check howamidoing to see whether you got the checkpoint point, and to submit a regrade request if you didn't
- Final project (Scheme) released today!
  - We strongly recommend starting as early as possible—this is a relatively long and challenging project, and it's common for students to get stuck; you'll have better access to office hours if you're not going on the day of the deadline!!
  - There are *two* checkpoints for Scheme—the first one is due Tuesday (8/2)
  - Along with the Scheme project, there's also a challenge version if you're into that idk
  - There's also an art contest!! You can submit a recursive art piece done in Scheme, and it will get put in our gallery for fellow students to view and vote on! This isn't worth any points, but the winners (might) get a fun prize 🤖

# Regular Expressions

# Declarative programming (again!)

A **regular expression** (also: **regex**) is a sequence of characters that specifies a **pattern**, usually utilized by string searching algorithms

Regular expressions, as they are typically used, are a great example of **declarative programming**—we can use a regular expression to specify a pattern that we want to find within a string, and then we give that pattern to a search algorithm to find the pattern for us

RegEx is *not* a programming language. It's a standard that is implemented within many modern programming languages (including Python!), because string search is a common problem

# Motivation

Imagine we want to solve any of these problems:

- Extract all the dates from a piece of text
- Find all the URLs in an email so we can convert them to hyperlinks
- Clean up code you wrote by removing trailing whitespace from the end of every line

All of these problems involve finding substrings in a longer string that fit a certain pattern, but doing exact string matching won't work

RegEx allows us to systematically describe patterns for these kinds of problems, that we can then pass off to pre-implemented algorithms

Of all the topics we learn in this course, RegEx is probably the one I personally use most in my life

## Matching exact strings

Most characters in RegEx will match exactly the characters as they appear in the expression

Expression: `abcabc`

Fully matched by: `abcabc`

Not fully matched by: `abdabc`, `abcab`

However, some characters in RegEx have reserved meanings, and so must be escaped with the backslash character if

you're interested in the character itself: `\` `(` `)` `[` `]` `{` `}` `+`  
`*` `?` `|` `$` `^` `.`

Expression: `\{abc\}`

Fully matched by: `{abc}`

# The dot character

In RegEx, the . character is a reserved character that will match any single character that is not a new line

Expression: `.a.a.a`

Fully matched by: `banana`, `aaaaaa`, `+a-a!a`

# Character classes

Character classes match any of a set of characters—one instance of a character class will match exactly one character

Expression: `[ab]c[ab]c`

Fully matched by: `acac`, `bcac`, `acbc`, `bcbc`

Not fully matched by: `aacac`, `accc`

Expression: `[a-z][0-9]`

Fully matched by: `a0`, `b8`, `z0`, `g5`



# Character classes

We have some shorthands for common character classes

- `.` matches any non-newline character
- `\d` matches digits, equivalent to `[0-9]`
- `\w` matches "word characters", equivalent to `[A-Za-z0-9_]`
- `\s` matches whitespace characters (spaces/tabs/line breaks)
- `[^]` matches any character *except* whatever comes after `^`
- `\D` matches any *non*-digit character (opposite of `\d`)  
`\s` and `\w` also have `\S` and `\W` as opposites

# Character classes: Examples

Expression: `[^ab]c`

Fully matched by: `cc`, `zc`, `!c`

Not fully matched by: `ac`, `bc`

Expression: `\(\d\d\d\) \d\d\d-\d\d\d\d`

Fully matched by: `(951) 262-3062`

# Quantifiers

Quantifiers allow us to specify multiple occurrences of the same character or character class

`a*` zero or more occurrences of `a`

`a+` one or more occurrences of `a`

`a?` zero or one occurrences of `a`

`a{2}` two occurrences of `a`

`a{2, 4}` two, three, or four occurrences of `a`

`a{2, }` at least two occurrences of `a`

In my experience, of everything you see in this lecture, `+` and `*` are the ones I've used the most

# Quantifiers: Examples

Expression: `a*b`

Fully matched by: `b`, `ab`, `aaaaaaaaaaaaab`

Expression: `a+b`

Fully matched by: `ab`, `aaaaaaaaaaaaab`

Not fully matched by: `b`

Expression: `\( \d{3} ) \d{3}-\d{4}`

Fully matched by: `( 951 ) 262-3062`

# Combining patterns

The pipe `|` operator matches either the expression on its left or its right

Expression: `\d+ | Inf`

Fully matched by: `78947892`, `Inf`

Not fully matched by: `78947892Inf`

You can also use parentheses `( )` to group expressions

Expression: `(<3)+`

Fully matched by: `<3`, `<3<3<3<3<3`

# Anchors

Anchors are unique in that they don't match characters—instead, they match positions in a string where an expression could land

`^` matches the beginning of a string

`$` matches the end of a string

`\b` matches a "word boundary" (whitespace, punctuation)

# Anchors: Examples

Expression: `^aw+`

Matched by: `aww` `aww`

Expression: `\w+y$`

Matched by: `stay` `stay`

Expression: `\w+e\b`

Fully matched by: `broken` `bridge` `team`

# Regular Expressions in Python



# The re module

Python has a module called `re` that supports regular expressions for strings!

This is common—many programming languages have some level of built-in support for regular expressions

```
>>> import re
```

```
>>> bool(re.search(r"\d+", "123 peeps"))
```

```
True
```

```
>>> bool(re.search(r"\d+", "So many peeps"))
```

```
False
```

# Raw strings

Python has escape characters built in to string evaluation, such as the newline character

```
>>> print("hello\nthere!")  
hello  
there!
```

Regular expressions also use the backslash character, but we don't want the Python interpreter to treat these as Pythonic escape characters. Raw strings allow us to use backslashes without worrying about this

```
>>> print(r"hello\nthere!")  
hello\nthere!
```

Evaluation rules for raw strings are confusing, and we won't test you on them—the important thing for you to know is that if you're writing a regular expression, you should use a raw string rather than a regular string

# Match objects

The `re` module has methods that attempt to match a pattern to a string—if they find a match, they'll return a `Match` object, and if they don't, they'll return `None`

`re.search(<pattern>, <string>)`

Returns a `Match` object representing the first occurrence of `<pattern>` in `<string>`

`re.fullmatch(<pattern>, <string>)`

Returns a `Match` object, requiring that `<pattern>` entirely match `<string>`

`re.match(<pattern>, <string>)`

Returns a `Match` object, requiring that `<string>` must start with a substring that matches `<pattern>`

## Match objects: Examples

```
>>> x = "This string contains 35 characters."  
>>> mat = re.search(r'\d+', x)  
>>> mat  
<re.Match object ...>  
>>> mat.group(0)  
'35'  
>>> mat2 = re.search(r'\d{3,}', x)  
>>> mat2  
>>> # Returned None :0
```

# Capturing groups

When we use parentheses to group sub-expressions, they define **capture groups** that we can then access individually

```
>>> x = "There were 12 pence in a shilling and 20  
shillings in a pound."  
>>> mat = re.search(r'(\d+)[a-z\s]+(\d+)', x)  
>>> mat.group(0)  
'12 pence in a shilling and 20'  
>>> mat.group(1)  
'12'  
>>> mat.group(2)  
'20'  
>>> mat.groups()  
('12', '20')
```

Capturing groups are super cool and helpful, but we won't test you on them in this class

## Other re functions

These are functions from the `re` module that don't return match objects

`re.findall(<pattern>, <string>)`

Returns a list of all substrings within `<string>` that match `<pattern>`, read from left to right

`re.sub(<pattern>, <repl>, <string>)`

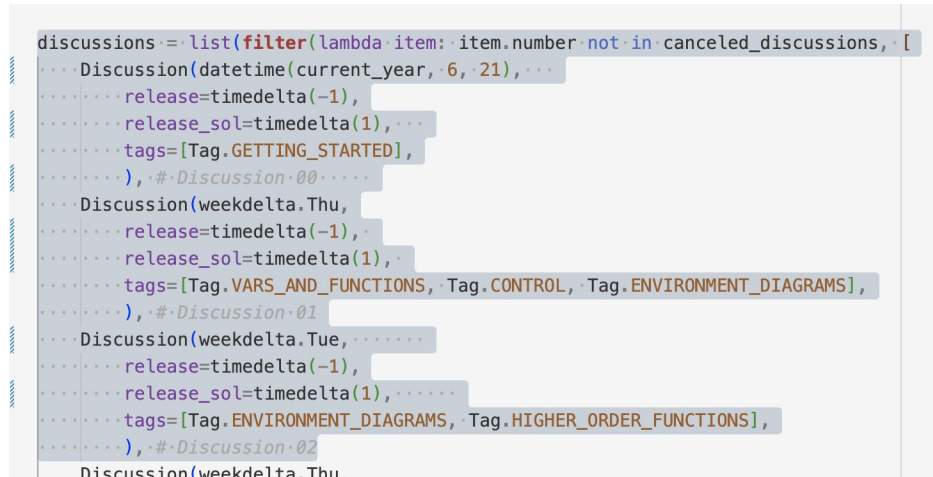
Returns `<string>`, but with all instances of `<pattern>` replaced with `<repl>`

Examples!

# Trimming whitespace

This is based on something real that I had to do lol

For formatting reasons, programmers tend to not like whitespace on the end of lines



```
discussions = list(filter(lambda item: item.number not in canceled_discussions, [
    Discussion(datetime(current_year, 6, 21),
        release=timedelta(-1),
        release_sol=timedelta(1),
        tags=[Tag.GETTING_STARTED],
    ), #Discussion·00
    Discussion(weekdelta.Thu,
        release=timedelta(-1),
        release_sol=timedelta(1),
        tags=[Tag.VARS_AND_FUNCTIONS, Tag.CONTROL, Tag.ENVIRONMENT_DIAGRAMS],
    ), #Discussion·01
    Discussion(weekdelta.Tue,
        release=timedelta(-1),
        release_sol=timedelta(1),
        tags=[Tag.ENVIRONMENT_DIAGRAMS, Tag.HIGHER_ORDER_FUNCTIONS],
    ), #Discussion·02
    Discussion(weekdelta.Thu,
```

Our website base actually has a built-in check that won't let you commit edits if there's trailing whitespace, which is the source of much pain for our content team



# Trimming whitespace

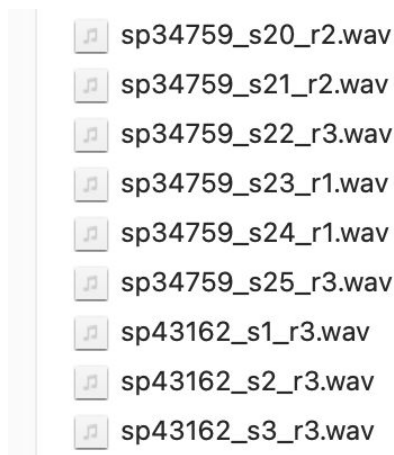
```
import re

in_f = open('example.txt', 'r')
lines = in_f.readlines()
# ^ reads in lines of a file as a list of strings
lines = [re.sub(r'\s+$', '', l) for l in lines]
out_f = open('fixed.txt', 'w')
out_f.writelines(['\n' + l for l in lines])
in_f.close()
out_f.close()
```

This is a script that trims whitespace, but I usually do it with find + replace in VS Code!

# Batch processing files

This is also a real example from my thesis! I had a bunch of files that looked like this:



Every file name is of the format

`sp<speaker_id>_s<sentence_id>_r<rep_#>.wav`

Where that file is a recording of speaker <speaker\_id> reading sentence <sentence\_id>, and it's their <rep\_#> iteration reading it (which doesn't matter very much)

# Batch processing files

```
import glob
import re
from parselmouth import Sound

wav_files = glob.glob("*.wav")
# ^ generates a list of all file names that end in .wav
sounds_by_id = {}
for w in wav_files:
    mat = re.fullmatch(r"sp(\d{5})_s(\d{1,2})_r\d\.wav", w)
    assert mat is not None, "Must match file format"
    speaker_id, sentence_id = mat.groups()
    if speaker_id not in files_by_id:
        files_by_id[speaker_id] = {}
    sounds_by_id[speaker_id][sentence_id] = Sound(w)
```

# Formal Language Theory

## Center embedding

Try to write a regular expression to match these strings:

b

abc

aabcc

aaabccc

aaaaaaaaaabcccccccccc

I.e., there are  $n$  a's, then one b, and then exactly  $n$  c's

# Context-free grammars (CFGs)

That problem is actually technically impossible, at least with regex

You can formally prove it, but you need to use some math that we're not really going to talk about in this class

Basically, regular expressions don't have "memory"—if I match 10 instances of the letter a, regular expressions don't have a way to remember that so that it can then match exactly 10 instances of the letter c

We can, however, do this with a different kind of pattern called a **context-free grammar**:

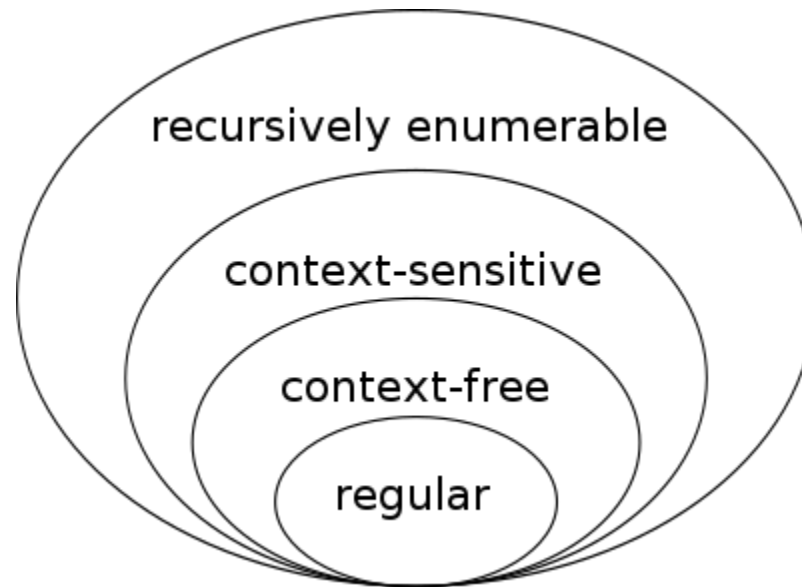
START ==> ABC

ABC ==> "a" + ABC + "c"

ABC ==> "b"

# The Chomsky Hierarchy

The Chomsky Hierarchy is an idea in **formal language** theory that we can use to classify what are called **formal grammars**



At the very bottom are **regular grammars**, like regular expressions, and context-free grammars are one step up from these

At the top are **recursively enumerable grammars**, which can fully model the behavior of a computer

# BNF

Although you need a recursively enumerable grammar to fully model computation, you can specify syntax for programming languages using a CFG!

**Backus-Naur Form** is a schema designed specifically for describing the syntax of programming languages using context-free grammars

BNF has been taught formally in previous iterations of this class, but it's out of scope this semester dw



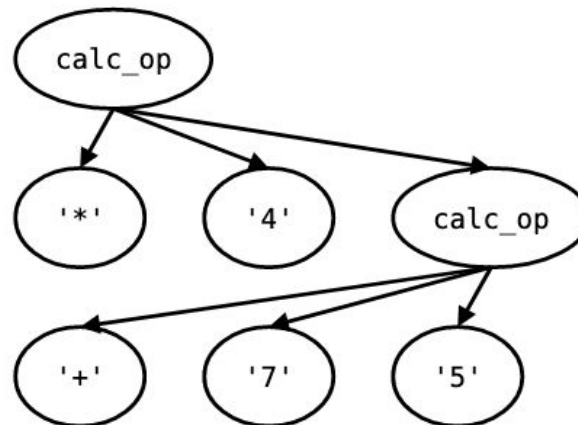
## BNF for (part of) Calculator

?start: calc\_expr

?calc\_expr: NUMBER | calc\_op

calc\_op: "(" OPERATOR calc\_expr\* ")"

OPERATOR: "+" | "-" | "\*" | "/"



If I make a really good CFG for the Scheme language, I can actually pass a BNF grammar into an algorithm to make my parser, which would have saved you a lot of work in Lab 9

# BNF for (part of) English

?start: sentence

sentence: noun\_phrase verb

noun: NOUN

noun\_phrase: article noun

article : | ARTICLE

verb: VERB

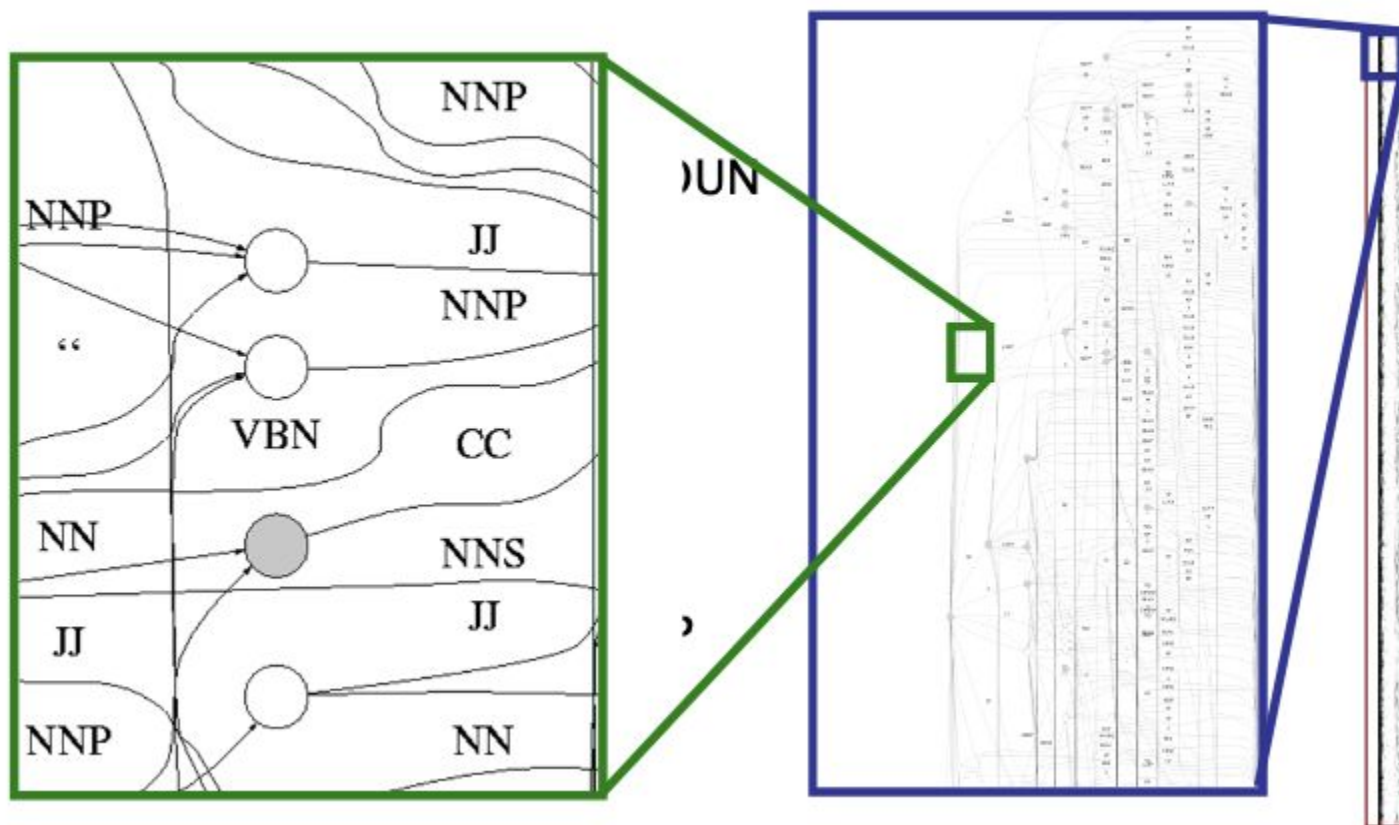
NOUN: "horse" | "dog" | "hamster"

ARTICLE: "a" | "the"

VERB: "stands" | "walks" | "jumps"

%ignore /\s+/

# Really big CFGs :0



(From CS 288 lecture slides)

# Review

Regular expressions are a way to define patterns that allow us to match strings in specified ways

Python has an `re` module that you can use to employ regular expressions in the language

You can't do everything with regex—there are, however, more powerful grammars that can specify more complex forms

In this class, you only need to worry about a couple of things:

- How to write a regular expression to match a given pattern
- How to read a regex and know whether it matches a string
- How to read Python code that uses regular expressions

## Tips

*Do not memorize everything*—have the docs in front of you, and as you use them with practice you'll get comfortable finding where things are

Regex can be really hard to debug—make sure you're being methodical, and breaking a bigger problem up into smaller pieces (just like any other kind of programming!) is often a great strategy

If you use Regex in the real world, be wary of long and complicated regular expressions—these can be hard to read for other developers, and often can have sinister bugs or edge cases

Also, keep in mind that there's no guarantee of any level of efficiency with pre-packaged Regex algorithms