COMP3420 Lesson 7

Greg Baker

2023-04-03

Contents

1 Introduction

Who is Greg Baker?

Figure 1: Very out-of-date photograph

- Ex-Google, ex-Atlassian, ex-CSIRO, former CTO of Daisee and Pixc
- Author of 6 books
- LinkedIn `https://www.linkedin.com/in/solresol`
- My research is in using number theory to make natural language processing work better on endangered and rare languages
- Fun fact: I proposed to my wife after we had been going out for 10 days. Married 27 years now.

Office hours

- Happy to answer questions about course, careers, jobs, startups, bootcamps
- After this class I'll wait around nobody is around (for up to an hour)
- If we get kicked out of SCO T5, then we'll go to the Ada Lovelace room
- ...or the food court if we're really late

Why are you doing this course?

- Do you want to start a start-up?
  - Vision or language?
- Learn about Chat GPT technology
- Interested in languages; e.g. linguistics major?
- You needed some credit points and this fit into your schedule?

What languages do you speak/read?

I'll try to come up with examples in relevant languages:

- European languages
- Chinese/other character-based writing

- Something with an exotic alphabet
- Ancient languages
- English only

Policy on use of Chat GPT

- This is the only class where playing with GPT is encouraged!
- I'll try to set labs that have some generated code and some "do this yourself" parts
- Also look at tabnine, Github Copilot for code completion specifically.
- LLMs to compare: Anthropic, Baidu*, Bing AI, GPT-3.5, GPT-4.0, Google PaLM (if it is ever released)

* I'm really interested to hear your experiences with this.

Policy on use of Chat GPT

- If you generate some code, do a git add and git commit afterwards
- Use "`[gpt4] your prompt here...`" as the commit message
- Make sure you can identify which commits are your work

Textbook
Speech and Language Processing, Jurafsky and Martin, Jan 2023 edition https://web.stanford.edu/~jurafsky/slp3/ed3book_jan72023.pdf

2 Course Outline

Course Outline

Apr 3 Unicode and Regular Expressions (Chapter 2)

Apr 24 Turning words into vectors (Chapter 14.1)

May 1 Explainable methods (Chapter 5)

May 8 Parts of speech and named entities (Chapter 8)

May 15 Embeddings (Chapter 6)

May 22 Chat GPT and large language models (Chapter 10.1, 10.2)

May 29 Review

Alternate view of the Course

Apr 3 Unicode and Regular Expressions 1950 - 1980 and their impact now

Apr 24 Turning words into vectors 1970 - 1990

May 1 Machine learning classifiers 1980 - 2000

May 8 Parts of speech and named entities 1990 - 2010

May 15 Embeddings 2000 - 2022

May 22 Chat GPT and large language models 2022-2023

May 29 Review

Yet Another Alternate View of the Course

Apr 3 Unicode and Regular Expressions Run a microcontroller for a millisecond

Apr 24 Turning words into vectors Run your phone for a second

May 1 Machine learning classifiers Run your laptop for a minute

May 8 Parts of speech and named entities Run your CPU hot for hours

May 15 Embeddings Run your GPU hot for days

May 22 Chat GPT and large language models Bake-the-planet energy consumption

One More Alternate View of the Course

Apr 3 Unicode and Regular Expressions What you do first on every language problem

Apr 24 Turning words into vectors When your first approach doesn't work

May 1 Machine learning classifiers Get smarter automatically and be explainable

May 8 Parts of speech and named entities Try something completely different if nothing else is working

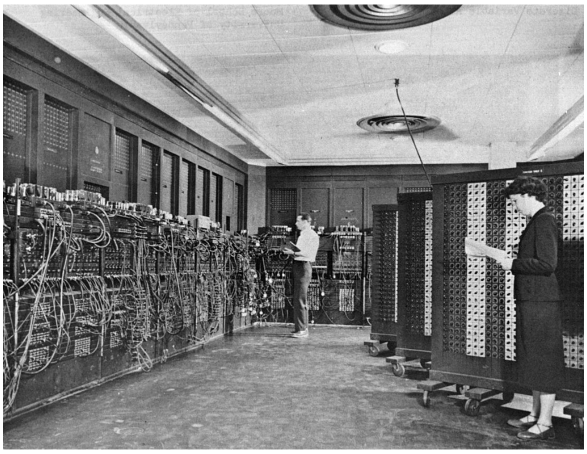
May 15 Embeddings Let's throw everything we have at the problem

May 22 Chat GPT and large language models Wave a magic wand and pay the $$$

3 Origins of Text Encodings

Very early days

- Every system had its own way of encoding text
- Portability of data between computers not considered much
- Natural language processing was "translation from Russian"



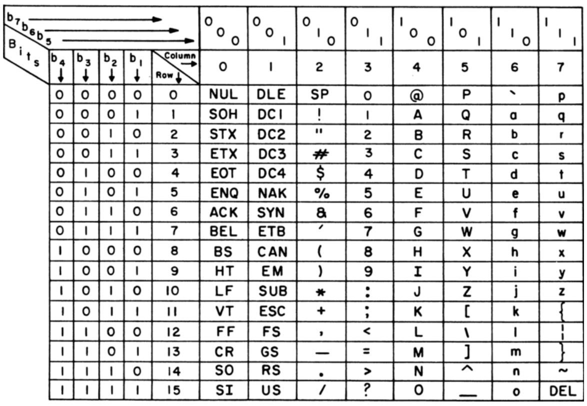
Dartmouth Time-Sharing System

https://www.youtube.com/watch?v=WYPNjSoDrqw

- Hugely influential in how we do computing today
- Tiny reference in that video to someone doing something with Latin (earliest reference I can find to AI with text that wasn't translation)
- Connected *teleprinters* (TTY) to a computer
- TTYs used the same encoding as telegrams

ASCII: (1961–1963)

- Purpose: Standardize character encoding for electronic communication (including computers)
- Upper case, lower case, punctuation, everything you need to write a text document in the USA!
- Record and data separators (that no-one uses)
- 7-bit storage: a compromise between 6-bit and 8-bit bytes?



b7 b6 b5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b4	b3	b2	b1	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	—	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

IBM (1964): The start of the mess

- IBM: We think each company should have their own standards!
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
- Based on punch cards
- `ord('z') - ord('a') == 41` (WTF?)
- *Still* how z-Series mainframes and AS/400s encode text
- Interested in working a job at bank or insurance company? Think about the encoding of transaction data, customer names, …

Use the `dd` program on Linux to convert, or convert on the mainframe.

The golden rule of text

When you have a stream of bytes, and you don't know how it was encoded, you have a useless stream of bytes.

If you *assume* an encoding, it will work fine for a while and then suddenly fails with weird error messages when you hit an impossible-in-that-encoding character.

No encoding? Code's exploding.

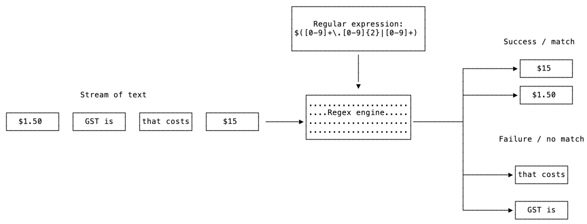
The story so far

- The date is somewhere around the late 1960s.
- Personal computers don't exist yet.
- Other than IBM, nearly everyone else is using ASCII.
- The CDC 7600 supercomputer ran at 36MHz, had 64k of memory, and rented for around $100,000 per month.
- You want to create a chatbot, classify some text, extract dates…

4 Traditional Regular Expressions

Introduction to Regular Expressions

- Invented by Stephen Kleene in the 1950s
- Purpose: to describe patterns in strings and search for specific patterns in text
- Tiny overlap with COMP3000, because a tokenizer for a compiler will usually uses regexes



Why would I care about this ancient history?

- They are *fast* to process large amounts of text. Billions of sentences? No problem.
- They are used in lots of programs: `grep`, `sed`, almost every text editor, Microsoft Word search, (many, many more) ...
- Almost every programming language includes them in their standard library (or deeper in the language): SQL, JavaScript, Python, Scala, Java, ...
- We use them as a pre-processing step in many language tasks.
- You are studying at a university. You can learn about things that are of purely academic interest. `;-)`

Basics

- An ordinary character (letters, numbers, spaces, most punctuation) matches that character
- Dot (`.`) matches any single character

`d.g`	matches	`dog`
`d.g`	matches	`dig`
`d.g`	matches	`d.g`
`d.g`	does not match	`dg`
`d.g`	does not match	`doug`
`d.g`	does not match	`gdb`

Repeaters

- `?` means “that last thing was optional”
- `*` means “zero or more of that last thing”
- *Later we'll see many more, but that's all there was in the 1970s*

`do?g`	matches	`dog`
`do?g`	matches	`dg`
`do?g`	does not match	`doog`
`do?g`	does not match	`dig`
`do*g`	matches	`dg`
`do*g`	matches	`dog`
`do*g`	matches	`doooog`
`d.*g`	matches	`drug`
`d.*g`	matches	`dddggg`

Alternates

- `[abc]` means “a letter ’a’, a letter ’b’ or a letter ’c”’
- `[âbc]` means “one letter, as long as it isn’t ’a’, ’b’ or ’c’”
- `[a-z]` means “any letter between ’a’ and ’z’” (in whatever encoding the regex is expecting)
- This counts as one “thing” for repeaters
- *Later we'll see more, but that's all there was in the 1970s*

`[a-z]*`	matches	`dog`
`[a-z]*`	does not match	`abc123`
`[0-9]*`	matches	(any integer)
`[a-f0-9]`	matches	(any hexadecimal number)

Anchors

In Python we can `.search()` (anywhere in the string) or `.match()` (start of the string) or `.fullmatch()` (whole string).

If we are *searching* then these might make sense to use:

`^` means “the start of a string”

`$` means “the end of a string”

Later we’ll see more, but that’s all there was in the 1970s

Searching for `^[a-z][a-z]*`	succeeds for	`dog`
Searching for `^[a-z][a-z]*`	succeeds for	`cat123`
Searching for `^[a-z][a-z]*`	fails for	`123cat`
Searching for `^[a-z][a-z]*`	succeeds for	`cat123`
Searching for `[a-z]*$`	succeeds for	`cat123`

(The last one: because “zero or more” lower case letters, followed by end of string matches)

GPT-3.5 can’t explain this properly

Why does a regex search for `[a-z]*$` succeed for `cat123`?

The regular expression [a-z]$ matches any string that ends with zero or more lowercase letters. ✓ In the case of the string "cat123", the regular expression matches the substring "cat" because it consists entirely of lowercase letters. ?? The remaining characters "123" do not match the regular expression, ✓ but since the regular expression only requires matching at the end of the string X, the overall match succeeds. Therefore, the regex search for [a-z]$ succeeds for "cat123".

GPT-4.0 can explain it, though

The regular expression `[a-z]*$` matches zero or more lowercase letters at the end of a string. Let’s break it down:

- `[a-z]`: This part of the regex matches any lowercase letter from `a` to `z`.
- `*`: The asterisk is a quantifier that means "match the preceding element (in this case, `[a-z]`) zero or more times."
- `$`: This is an anchor that asserts the position at the end of the string.

When you apply this regex to the string `"cat123"`, it does find a match, but the match is an empty string. Here’s why:

- The `[a-z]*` part of the regex can match zero or more lowercase letters. In the case of `"cat123"`, there are no lowercase letters at the end of the string, so it matches zero occurrences of `[a-z]`.
- The `$` part of the regex asserts the position at the end of the string, and since there are no lowercase letters at the end of `"cat123"`, the match is an empty string right before the end-of-string position.

So, the regex `[a-z]*$` succeeds for `"cat123"` because it matches an empty string at the end of the input string.

Some mnemonics

- There are lots of stars (*) in the universe.
- Maybe you’ll ask a question (?)
- Pick one from the square [] that’s there.
- At the start of the day, you put on a hat (^)
- In the end, it’s just about $.
- A back(s)lash \ cancels anything special.

A first regex program

```python
#!/usr/bin/env python3
import re
money_matcher =
    re.compile('\$[0-9]+\.[0-9][0-9]')
while True:
    try:
        text = input('Text to check >> ')
    except EOFError:
        break
    matching = money_matcher.search(text)
    if matching is None:
        print("No money mentioned")
        continue
    span_start, span_end = matching.span()
    print("The first money reference is",
        text[span_start:span_end])
```

Just for reference, how it looks in Javascript

```javascript
const moneyMatcher =
    /\$[0-9][0-9]*\.[0-9][0-9]/;
text = "That will be $2.50";
matching = text.match(moneyMatcher);
if (matching === null) {
    console.log('No money mentioned');
    return;
}
starts_at = matching.index;
console.log(`The first money reference
    starts at character ${starts_at}`);
```

(We’ll keep using Python in this course)

Battling with Python strings

If you write an “ordinary” string, Python will do some replacements:

- `\\` - Backslash itself
- `\'` - Single quote
- `\"` - Double quote
- `\n` - Newline
- `\r` - Carriage return
- `\t` - Tab
- `\b` - Backspace
- `\f` - Formfeed
- `\ooo` - Octal value (replace `ooo` with the octal value)
- `\xhh` - Hex value (replace `hh` with the hex value)
- `\Uxxxxxxxx` - Unicode character with hex value (replace `xxxxxxxx` with the hex value of the unicode character you wnat)

```
print("The first line.\nThe second line
    of output\nSmiley face\U0001F603")
```

Matching a backslash at the end of line

Regex: `\\$`

Python version (ugly):

```
matcher = re.compile("\\\\$")
```

Python version (better)

```
matcher = re.compile(r"\\$")
```

r before a string means “raw”

Implementing Regular Expressions

Don’t. There are libraries for this.

Regexes can be turned into finite state machines.

Alternate technique is Brzozowski derivative

You won’t have to spell “Brzozowski” in the exam. I promise this slide won’t be in the exam at all in fact.

What we can do now (1970s edition)

Entity Extraction

- Find capitalised words (likely to be names / proper nouns)
- Find dates
- Find numbers and currency amounts

Human Language

- Find a singular or plural version of a regular nouns (`s?`)
- Find regular verbs in different tenses (`-ed`)
- Find many adverbs (`-ly`)

Computer Languages

- Recognise a valid variable or function name
- Recognise language keywords and syntax

5 Modern Text Encodings

What about the rest of Europe?

French é è ô

German ö

Spanish ñ

IBM/Microsoft code pages (1984)

The unused ASCII 8th bit allowed for the creation of extended ASCII character sets.

IBM and Microsoft worked together, and defined:

Code Page 1252 Most of western Europe and parts of Africa, except for Dutch and Slovene which were hackily-supported.

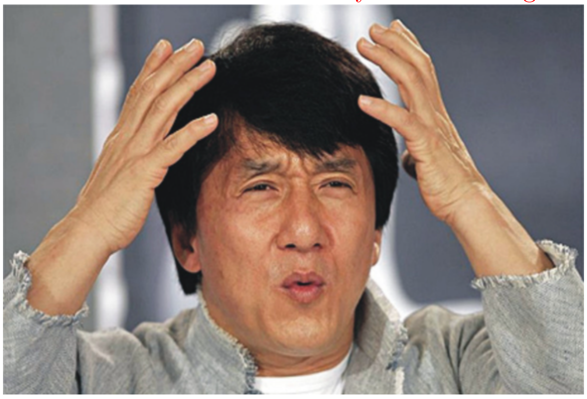
Code Page 1253 Greek

Code Page 1251 Cyrillic (e.g. Russian)

(Plus many more, for almost every language.)

Implications of Code Pages

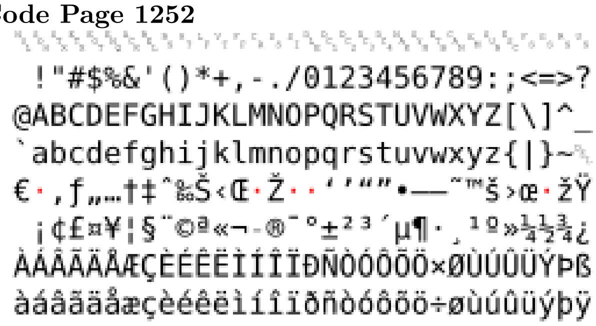
A Russian Word document could not be shared with a French user unless they wrote in English.



Mojibake

"the garbled text that is the result of text being decoded using an unintended character encoding"

Code Page 1252



ISO 8859-1

In 1988 the International Standards Organisation establised ISO 8859-1 "Latin-1" encoding.

Microsoft: We can out-mess you

- Microsoft: We think each company should have their own standards!
- We'll continue to use Code Page 1252, which has different characters to ISO-8859-1, but we'll call it ISO-8859-1 anyway!
- We'll make up own own open quotes and close quotes characters so that it affects *almost every document written*

Remember the golden rule of text

When you have a stream of bytes, and you don't know how it was encoded, you have a useless stream of bytes.

If you *believed* the encoding that Microsoft products told you, it would work fine for a while and then suddenly fail with weird error messages when you hit an impossible-in-that-encoding character.

Dodgy encoding? Code's exploding.

A little more Python background

More string prefixes:

f Format string, e.g. `f'1 + 2 = {1+2}'`

u Unicode string (same as without `u`)

b Sequence of bytes (unknown encoding)

r Raw string (don't interpret \)

Encoding / Decoding

```
x = b'\x5a\x40'
print(x.decode('ascii')) ;#
    prints Z@
print(x.decode('cp1252')) ;#
    prints Z@
y = 'Z@'
print(y.encode('ascii'))  ;#
    prints b'Z@'
print(y.encode('utf-16')) ;#
    prints b'\xff\xfeZ\x00@\x00'
```

Adventuring outside ASCII

```
x = b'\x93'
print(x.decode('ascii')) ;#
    exception
print(x.decode('cp1252')) ;# open
    double quotes
print(x.decode('iso-8859-1')) ;#
    nothing - control char
print(x.decode('latin-1')) ;#
    nothing - control char
print(x.decode('utf-16')) ;#
    exception
```

(Always throws `UnicodeDecodeError` for a failure, regardless of codec in use)

open

`open` takes many arguments

file The filename

mode e.g. `rb`, `wt`

buffering Line-based (1), fixed-block size, none (0).

encoding e.g. `ascii`, `latin-1`

errors How to handle decoding errors, e.g. `strict`, `ignore`, `replace`, `backslashreplace`

(There are many others)

Examples of using open

```
f = open('example.utf16', 'w',
    encoding='utf-16')
f.write("Hello world")
f.close()
g = open('example.utf16')
g.read()
```

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
  File "/Users/gregb/miniconda3/lib/python3.10/codecs.py", line 322, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte
```

More examples of open

```
with open('example.utf16', 'w',
    encoding='utf-16') as f:
     f.write("Hello world")
binary_file =
    open('/tmp/foo.utf16', 'rb')
binary_file.read()
```

```
b'\xff\xfeH\x00e\x00l\x00l\x00o\x00 \x00w\x00o\x00r\x00l\x00d\x00'
```

How the world responded to Microsoft's CP1252 mess

All right, whenever we see ISO-8859-1, we'll just assume it means CP1252.

International Standards Organisation

Can we do something about code pages?

Like, one **uni**versal **code** page for every language?

Something that Microsoft and IBM can't mess up?

Please?

Microsoft: UTF-16

"There are less than 65,536 different letters in the world. So we can encode everything in 16 bits."

Class polling question

How many different Chinese characters are there?

Varying-width encoding

Simple and common letters (e.g. ASCII characters) should take up fewer bytes than rarely-used symbols from exotic languages.

$\rightarrow$ The length of a string $\neq$ The number of letters

UTF-16

UTF-16 is *now* a varying-width encoding, negating the whole point of using 16-bit code points.

UTF-16 is the default on Windows systems (filenames, documents, network protocols etc.)

UTF-8

1993: Ken Thompson and Rob Pike proposed a new encoding called UTF-8.

"If we have to have varying-width encodings anyway, why not be compatible with ASCII, and generally work in 8-bits?"

UTF-8 is the default on Linux, OSX and is an optional component to install on Windows.

Fun fact: You can use Unicode in variable and function names in Python. e.g. `😄 += 1`

ASCII characters in UTF-8

- Explanation: ASCII characters are represented using a single byte in UTF-8
- Example: ASCII character "A" is encoded as `0x41` in UTF-8

An ASCII string is also a valid UTF-8 string

Two-byte characters in UTF-8

- Explanation: Characters in the range 128-2047 are represented using two bytes in UTF-8
- Byte layout:
  - First byte: 110xxxxx (where xxxxx is the character data)
  - Second byte: 10xxxxxx (where xxxxxx is the character data)
- Example: Character "é" is encoded as `0xC3 0xA9` in UTF-8

Three-byte characters in UTF-8

- Explanation: Characters in the range 2048-65535 are represented using three bytes in UTF-8
- Byte layout:

- First byte: 1110xxxx (where xxxx is the character data)
- Second byte: 10xxxxxx
- Third byte: 10xxxxxx

- Example: Mandaic letter "AB" ࡁ is encoded as `0xE0 0xA1 0x81` in UTF-8

Four-byte characters in UTF-8

- Explanation: Characters in the range 65536-1114111 are represented using four bytes in UTF-8
- Byte layout:
  - First byte: 11110xxx (where xxx is the character data)
  - Second byte: 10xxxxxx
  - Third byte: 10xxxxxx
  - Fourth byte: 10xxxxxx
- Example: Emoji 😄 (character 128516 = U+1F604) is encoded as `0xF0 0x9F 0x98 0x84`

Summary of text encoding

The golden rule: a stream of bytes is useless without knowing the encoding.

ASCII Simple, limited, upwardly compatible with UTF-8

EBCDIC Ancient IBM technology.

CP1252 A *code page* for Western European languages

CPxxxx Go and look up what the Microsoft code page is for your language: old PCs will still use it

ISO-8859-1/Latin-1 Closely related format

UTF-16 Modern Windows default, encodes for Unicode

UTF-8 Where the world will end up, encodes for Unicode

6 Perl-Compatible Regular Expressions

Repeaters

1984: Larry Wall releases the Perl language
New baseline for regexps

+ At least one (possibly more) of the previous thing

{4} Exactly 4 of the previous thing

{2,4} 2,3 or 4 of the previous thing

{2,} 2 or more of the previous thing

`do+g`	matches	`dog`
`do+g`	matches	`dooog`
`do+g`	does not match	`dg`
`d.{2,4}g`	matches	`doug`
`d.{2,4}g`	does not match	`dig`

Parentheses

Grouping `a(bc)*d` matches `abcd`, `ad` and `abcbcbcbcd` but not `abbd`

Alternation `(Jan|Feb|Mar)` matches those months

Capturing `([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})` matches and saves a year, month and date

Example of capturing

```python
#!/usr/bin/env python3
import re
date_finder = re.compile(
    '([0-9]{4})-([0-9]{1,2})-([0-9]{1,2})')
try:
    text = input('Text to search >> ')
    matching = date_finder.search(text)
    if matching is None:
        print("No dates mentioned")
    else:
        year = matching.group(1)
        month = matching.group(2)
        day = matching.group(3)
        print(f"Found {year=} {month=}
            {day=}")
except EOFError:
    pass
```

Character Class Shorthands

\d ASCII digits (like `[0-9]`)

\D Not an ascii digit (like `[^0-9]`)

\w Word-like (letters, numbers, underscore) (like `[a-zA-Z0-9_]`)

\w Not word-like (like `[^a-zA-Z0-9_]`)

\s Whitespace (e.g. space, tab, newline)

\S Not whitespace

\b Boundary between whitespace and not-whitespace (zero characters long)

\B Not a boundary between whitespace and not-whitespace (zero characters long)

Use of .split

```
#!/usr/bin/env python3
import re
splitter = re.compile(r'\b')
try:
    text = input('Text to split up >> ')
    words = splitter.split(text)
    print(f"Your words were: {';
        '.join(words)}")
except EOFError:
    pass
```

Unicode classes

```
pattern = re.compile(r'['
'\U0001f600-\U0001f64f'
'\U0001f300-\U0001f5ff'
'\U0001f680-\U0001f6ff'
'\U0001f1e0-\U0001f1ff'
']'
```

Use of .sub

```
#!/usr/bin/env python3
import re
smiley =
    re.compile(r'[\U0001F600-\U0001F610]')
try:
    text = input('Emoji replacer >> ')
    new_text = smiley.sub(';-)', text)
    print(new_text)
except EOFError:
    pass
```

Lookaround

(?=...) Must be followed by

(?!...) Must not be followed by

(?<!...) Must not be preceded by

Finding un-negated -ing words

```
#!/usr/bin/env python3
import re
pattern = re.compile(
    r'(?<!not )\b(\w+ing)\b')
try:
    text = input('Positive gerunds >> ')
    if matching := pattern.search(text):
        print(matching.group(1))
except EOFError:
    pass
```

Summary

- Text = bytes + encoding
- Regular expressions are fast
- Regexp syntax: classic + PCRE

Bye!

Next week:

Turning words into vectors (Chapter 14.1)

Good luck with your assignment, and enjoy your break.

7 Regex poetry

I asked ChatGPT 4.0 to wrote a poem to help remember basic regex syntax.

7.1 Attempt 1

small

Prompt: `Write a poem that helps students remember regex syntax. Include [], *, ?, ^ and $`

In a world of strings and text we dwell,
Regex patterns we must learn to tell,
A poem to guide, a friend to know,
To master syntax, let's begin this show.

With square brackets, '[ ]', we commence,
A character set they represent,
One letter, number, symbol to choose,
The power of regex you shall infuse.

*

An asterisk, '*', stands steadfast and tall,
Matches zero or more, that's not all,
Preceding character, a loyal mate,
Together they replicate.

The question mark, '?', a symbol we ponder,
An optional character, it helps us wander,
Matches zero or one, never more,
A flexible pattern we adore.

The caret, ”, denotes the start,
Anchoring patterns, a crucial part,
From the beginning, it takes its cue,
Matching strings, steadfast and true.

And finally, the dollar sign, '$', we reach,
The end of line, it does beseech,
Anchored once more, but at the close,
Regex mastery, you now propose.

With regex patterns in your grasp,
Text manipulation within your clasp,
Rhymes to remember, a guide at hand,
In the realm of strings, you now command.

7.2 Attempt 2

Prompt: `Write a cute rhyming poem to help students remember regular expression syntax. Use the following mnenomnics as a starting point:`

- `There are lots of stars (*) in the universe.`
- `Maybe you'll ask a question (?)`
- `Pick one from the square [] that's there.`
- `At the start of the day, you put on a hat (ˆ)`
- `In the end, it's just about $.`
- `A back(s)lash \ cancels anything special.`

In the universe, stars (*) twinkle so bright,
They match many times, in patterns of light.
When you're unsure, just ask a question (?),
It may appear once or not, a subtle suggestion.

Pick one from the square [] that's near,
A range of options, their presence to cheer.
At the start of the day, you put on a hat (ˆ),
Marking beginnings with a tip of that.

In the end, it's just about $, they say,
Matching the finish, the close of the day.
A back(s)lash brings calm to the scene,
Cancels the special, keeping it serene.

So remember these rhymes, these little tricks,
For regular expressions, they'll be your picks.
With their help, you'll navigate this code land,
And master the patterns with a confident hand.