

CHAPTER

2

Words and Tokens

User: I need some help, that much seems certain.
ELIZA: WHAT WOULD IT MEAN TO YOU IF YOU GOT SOME HELP
User: Perhaps I could learn to get along with my mother.
ELIZA: TELL ME MORE ABOUT YOUR FAMILY
User: My mother takes care of me.
ELIZA: WHO ELSE IN YOUR FAMILY TAKES CARE OF YOU
User: My father.
ELIZA: YOUR FATHER
User: You are like my father in some ways.

Weizenbaum (1966)

ELIZA The dialogue above is from **ELIZA**, an early natural language processing system that could carry on a limited conversation with a user by imitating the responses of a Rogerian psychotherapist (Weizenbaum, 1966). ELIZA is a surprisingly simple program that uses pattern matching on words to recognize phrases like “I need X” and change the words into suitable outputs like “What would it mean to you if you got X?”. ELIZA’s mimicry of human conversation, while very crude by modern standards, was remarkably successful: many people who interacted with ELIZA came to believe that it really *understood* them. As a result, this work led researchers to first think about the impacts of chatbots on their users (Weizenbaum, 1976).

tokenization Of course modern chatbots don’t use the simple pattern-based mimicry that ELIZA pioneered. Yet the pattern-based approach to words instantiated in ELIZA is still relevant today in the context of **tokenization**, the task of separating out or **tokenizing** words and word parts from running text. Tokenization, the first step in modern NLP, includes pattern-based approaches that date back to ELIZA.

To understand tokenization we first need to ask: What is a word? Is *um* a word? What about *New York*? Is the nature of words similar across languages? Some languages, like Vietnamese or Cantonese, have very short words while others, like Turkish, have very long words. We also need to think about how to represent words in terms of **characters**. We’ll introduce **Unicode**, the modern system for representing characters, and the **UTF-8** text encoding. And we’ll introduce the **morpheme**, the meaningful subpart of words (like the morpheme *-er* in the word *longer*)

BPE The standard way to tokenize text is to use the input characters to guide us. So once we understand the possible subparts of words, we’ll introduce the standard **Byte-Pair Encoding (BPE)** algorithm that automatically breaks up input text into tokens. This algorithm uses simple statistics of letter sequences to induce a vocabulary of subword tokens. All tokenization systems also depend on **regular expressions** as a processing step. The regular expression is a language for formally specifying and manipulating text strings, an important tool in all modern NLP systems. We’ll introduce regular expressions and show examples of their use

regular expressions

Finally, we’ll introduce a metric called **edit distance** that measures how similar two words or strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other. Edit distance plays a role in NLP whenever we need compare two words or strings, for example in the crucial **word error rate** metric for automatic speech recognition.

2.1 Words

How many words are in the following sentence?

They picnicked by the pool, then lay back on the grass and looked at the stars.

This sentence has 16 words if we don't count punctuation as words, 18 if we count punctuation. Whether we treat period (“.”), comma (“,”), and so on as words depends on the task. Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks). Large language models generally count punctuation as separate words.

utterance Spoken language introduces other complications with regard to defining words. What about this utterance from a spoken conversation? (**Utterance** is the technical linguistic term for the spoken correlate of a sentence).

I do uh main- mainly business data processing

disfluency fragment filled pause This utterance has two kinds of **disfluencies**. The broken-off word *main-* is called a **fragment**. Words like *uh* and *um* are called **fillers** or **filled pauses**. Should we consider these to be words? Again, it depends on the application. If we are building a speech transcription system, we might want to eventually strip out the disfluencies. But we also sometimes keep disfluencies around. Disfluencies like *uh* or *um* are actually helpful in speech recognition in predicting the upcoming word, because they may signal that the speaker is restarting the clause or idea, and so for speech recognition they are treated as regular words. Because different people use different disfluencies they can also be a cue to speaker identification. In fact [Clark and Fox Tree \(2002\)](#) showed that *uh* and *um* have different meanings in English. What do you think they are?

word type Perhaps most important, in thinking about what is a word, we need to distinguish two ways of talking about words that will be useful throughout the book. Word **types** are the number of distinct words in a corpus; if the set of words in the vocabulary is V , the number of types is the **vocabulary size** $|V|$. Word **instances** are the total number N of running words.¹ If we ignore punctuation, the picnic sentence has 14 types and 16 instances:

They picnicked by the pool, then lay back on the grass and looked at the stars.

We still have decisions to make! For example, should we consider a capitalized string (like *They*) and one that is uncapitalized (like *they*) to be the same word **type**? The answer is that it depends on the task! *They* and *they* might be lumped together as the same type in some tasks where we care less about the formatting, while for other tasks, capitalization is a useful feature and is retained. Sometimes we keep around two versions of a particular NLP model, one with capitalization and one without capitalization.

So far we have been talking about **orthographic words**: words based on our English writing system. But there are many other possible ways to define words. For example, while orthographically I'm is one word, grammatically it functions as two words: the subject pronoun I and the verb 'm, short for am.

¹ In earlier tradition, and occasionally still, you might see word instances referred to as word *tokens*, but we now try to reserve the word *token* instead to mean the output of subword tokenization algorithms.

| Corpus | Types = $ V $ | Instances = N |
|-------------------------------------|---------------|-----------------|
| Shakespeare | 31 thousand | 884 thousand |
| Brown corpus | 38 thousand | 1 million |
| Switchboard telephone conversations | 20 thousand | 2.4 million |
| COCA | 2 million | 440 million |
| Google n-grams | 13 million | 1 trillion |

Figure 2.1 Rough numbers of wordform types and instances for some English language corpora. The largest, the Google n-grams corpus, contains 13 million types, but this count only includes types appearing 40 or more times, so the true number would be much larger.

The distinctions get even harder to make once we start to think about other languages. For example the writing systems of languages like Chinese, Japanese, and Thai simply don't have orthographic words at all! That is, they don't use spaces to mark potential word-boundaries. In Chinese, for example, words are composed of characters (called **hanzi** in Chinese). Each character generally represents a single unit of meaning (called a **morpheme**, introduced below) and is pronounceable as a single syllable. Words are about 2.4 characters long on average. But since Chinese has no orthographic words, deciding what counts as a word in Chinese is complex. For example, consider the following sentence:

(2.1) 姚明进入总决赛 yáo míng jìn rù zǒng jué sài
 “Yao Ming reaches the finals”

As Chen et al. (2017) point out, this could be treated as 3 words (a definition of words called the ‘Chinese Treebank’ definition, in which Chinese names (family name followed by personal names) are treated as a single word):

(2.2) 姚明 进入 总决赛
 YaoMing reaches finals

But the same sentence could be treated as 5 words (‘Peking University’ standard), in which names are separated into their own units and some adjectives appear as distinct words:

(2.3) 姚 明 进 入 总 决 赛
 Yao Ming reaches overall finals

Finally, it is possible in Chinese simply to ignore words altogether and use characters as the basic elements, treating the sentence as a series of 7 characters, which works pretty well for Chinese since characters are at a reasonable semantic level for most applications (Li et al., 2019):

(2.4) 姚 明 进 入 总 决 赛
 Yao Ming enter enter overall decision game

But that method doesn't work for Japanese and Thai, where the individual character is too small a unit.

These issues with defining words makes it hard to use words as the basis for tokenizing text in NLP across languages.

But there's another problem with words. There are too many of them!!! How many words are there in English? When we speak about the number of words in the language, we are generally referring to word types. Fig. 2.1 shows the rough numbers of types and instances computed from some English corpora.

You will notice that the larger the corpora we look at, the more word types we find! That suggests that there is not a clear answer to how many words there are; the answer keeps growing as we see more data! We can see this fact mathematically

Herdan's Law
Heaps' Law

because the relationship between the number of types $|V|$ and number of instances N is called **Herdan's Law** (Herdan, 1960) or **Heaps' Law** (Heaps, 1978) after its discoverers (in linguistics and information retrieval respectively). It is shown in Eq. 2.5, where k and β are positive constants, and $0 < \beta < 1$.

$$|V| = kN^\beta \quad (2.5)$$

The value of β depends on the corpus size and the genre; numbers from 0.44 to 0.56 or even higher have often been reported. Roughly we can say that the vocabulary size for a text goes up a little faster than the square root of its length in words.

function words

content words

There are also variants of the law, which capture the fact that we can distinguish roughly two classes of words. One is **function words**, the grammatical words like English *a* and *of*, that tend not to grow indefinitely (a language tends to have a fixed number of these). The other is **content words**: nouns, adjectives and verbs that tend to have meanings about people and places and events. Nouns, and especially particular nouns like names and technical terms do tend to grow indefinitely. So models that are sensitive to this difference between function words and content words have one value of β for the initial part of the corpus where all words are still appearing, and then a second β afterwards for when only the content words are still appearing. Fig. 2.2 shows an example from Tria et al. (2018) showing two values of β (called γ in their figure) for Heaps law computed on the Gutenberg corpus of books. Note that

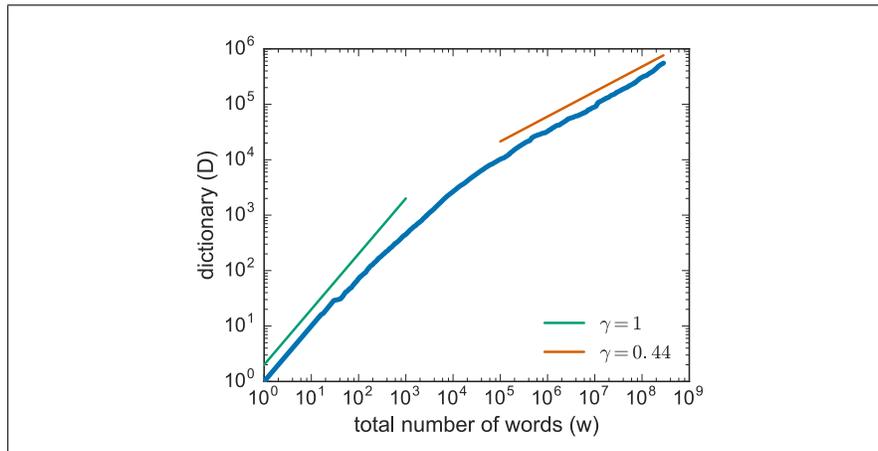


Figure 2.2 The thick blue line shows Vocabulary size $|V|$ (called D in their figure) as a function of text length (their w), computed on the Gutenberg corpus of publicly available books. Note that at the beginning of the corpus, we see both common and function words and the relationship between corpus size and vocabulary is roughly linear (green line, $\gamma = 1$). Later, after the function words have mainly appeared, the number of new words slows down and is closer to the square root of the corpus size. Figure from Tria et al. (2018).

The fact that words grow without end leads to a problem for any computational model. No matter how big our vocabulary, we will never have a vocabulary that captures all the possible words that might occur! That means that our computational model will constantly see **unknown words**: words that it has never seen before. This is a huge problem for machine learning models.

Because of these two problems (first, that many languages don't have orthographic words, and defining them post-hoc is challenging and second, that the number of words grows without bound), language models and other NLP models don't

tend to use words as their unit of processing. Instead, they use smaller units called **subwords** that can be recombined to model new words that our model has never seen before. To think about defining subwords, we first need to talk about units that are smaller than words; **morphemes** and **characters**.

2.2 Morphemes: Parts of Words

morphology
morpheme

Words have parts. At the level of characters, this is obvious. The word *cats* is composed of four characters, ‘c’, ‘a’, ‘t’, ‘s’. But this is also true at a more subtle level: words have components that themselves have coherent meanings. These components are called **morphemes**, and the study of morphemes is called **morphology**. A **morpheme** is a minimal meaning-bearing unit in a language. So, for example, the word *fox* consists of one morpheme (the morpheme *fox*) while the word *cats* consists of two: the morpheme *cat* and the morpheme *-s* that indicates plural.

Here’s a sentence in English segmented into morphemes with hyphens:

(2.6) Doc work-ed care-ful-ly wash-ing the glass-es

As we mentioned above, in Chinese, conveniently, the writing system is set up so that each character mainly describes a morpheme. Here’s a sentence in Mandarin Chinese with each morpheme character glossed, followed by the translation:

(2.7) 梅干菜 用清水泡软, 捞出后, 沥干
plum dry vegetable use clear water soak soft, remove out after, drip dry
切碎
chop fragment

Soak the preserved vegetable in water until soft, remove, drain, and chop

root
affix

We generally distinguish two broad classes of morphemes: **roots**—the central morpheme of the word, supplying the main meaning—and **affixes**—adding “additional” meanings of various kinds. In the English example above, for the word *worked*, *work* is a root and *-ed* is an affix; similarly for *glasses*, *glass* is a root and *-es* an affix.

inflectional
morphemes

Affixes themselves fall into two classes, or more correctly a continuum between two poles. At one end, **inflectional morphemes** are grammatical morphemes that tend to play a syntactic role, such as marking agreement. For example, English has the inflectional morpheme *-s* (or *-es*) for marking the **plural** on nouns and the inflectional morpheme *-ed* for marking the past tense on verbs. Inflectional morphemes tend to be productive and often obligatory and their meanings tend to be predictable.

derivational
morphemes

Derivational morphemes are more idiosyncratic in their application and meaning. Usually they apply only to a specific subclass of words and result in a word of a *different* grammatical class than the root, often with a meaning hard to predict exactly. In the example above, the word *care* (a noun) can be combined with the derivational affix *-full* to produce an adjective (*careful*), and another derivational affix *-ly* to result in an adverb (*carefully*).

clitic

There is another class of morphemes: **clitics**. A clitic is a morpheme that acts syntactically like a word but is reduced in form and attached (phonologically and sometimes orthographically) to another word. For example the English morpheme *’ve* in the word *I’ve* is a clitic; it has the grammatical meaning of the word *have*, but in form it cannot appear alone (you can’t just say the sentence “’ve”). The English possessive morpheme *’s* in the phrase *the teacher’s book* is a clitic. French definite

article *l'* in the word *l'opera* is a clitic, as are prepositions in Arabic like *b* ‘by/with’ and conjunctions like *w* ‘and’.

morphological
typology

The study of how languages vary in their morphology, i.e., how words break up into their parts, is called **morphological typology**. While morphologies of languages can differ along many dimensions, two dimensions are particularly relevant for computational word tokenization.

isolating

The first dimension is the **number of morphemes per word**. In some languages, like Vietnamese and Cantonese, each word on average has just over one morpheme. We call languages at this end of the scale **isolating** languages. For example each word in the following Cantonese sentence has one morpheme (and one syllable):

(2.8) *keoi5 waa6 cyun4 gwok3 zeoi3 daai6 gaan1 uk1 hai6 ni1 gaan1*
 he say entire country most big building house is this building
 “He said the biggest house in the country was this one”

synthetic
polysynthetic

Alternatively, in languages like Koryak, a Chukotko-Kamchatkan language spoken in the northern part of the Kamchatka peninsula in Russia, a single word may have very many morphemes, corresponding to a whole sentence in English (Arkadiev, 2020; Kurebito, 2017). We call languages toward this end of the scale **synthetic** languages, and the very end of the scale **polysynthetic** languages.

(2.9) *t-ə-nk'e-mejŋ-ə-jetemə-nni-k*
 1SG.S-E-midnight-big-E-yurt.cover-E-sew-1SG.S[PFV]
 “I sewed a lot of yurt covers in the middle of a night.”
 (Koryak, Chukotko-Kamchatkan, Russia; Kurebito (2017, 844))

Fig. 2.3 shows an early computation of morphemes per words on a few languages by the linguistic typologist Joseph Greenberg (1960).

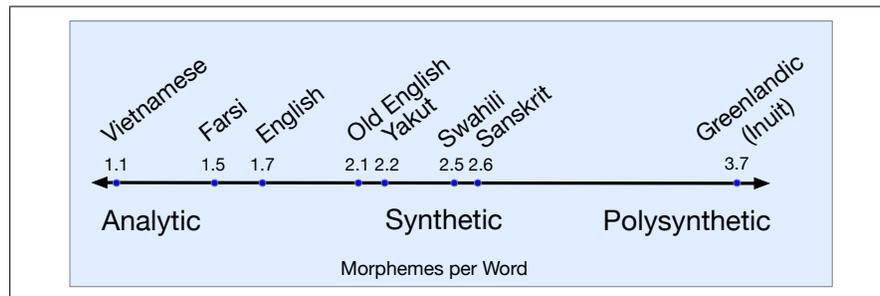


Figure 2.3 An early estimate of morphemes per word by Joseph Greenberg (1960).

agglutinative
fusion

The second dimension is the degree to which morphemes are easily segmentable, ranging from **agglutinative** languages like Turkish, in which morphemes have relatively clean boundaries, to **fusion** languages like Russian, in which a single affix may conflate multiple morphemes, like *-om* in the word *stolom* (table-SG-INSTR-DECL1), which fuses the distinct morphological categories instrumental, singular, and first declension.

The English *-s* suffix in *She reads the article* is an example of fusion, since the suffix means both third person singular but also means present tense, and there’s no way to divide up the meaning to different parts of the *-s*.

Although we have loosely talked about these properties (analytic, polysynthetic, fusional, agglutinative) as if they are properties of languages, in fact languages can make use of different morphological systems so it would be more accurate to talk about these as general tendencies.

Nonetheless, the fact morphemes can be hard to define, and that many languages can have complex morphemes that aren't easy to break up into pieces makes it very difficult to use morphemes as a standard for tokenization cross-lingually.

2.3 Unicode

Unicode Another option we could consider for tokenization is the level of the individual character. How do we even represent characters across languages and writing system? The **Unicode** standard is a method for representing text written using any character in any script of the languages of the world (including dead languages like Sumerian cuneiform, and invented languages like Klingon).

ASCII Let's start with a brief historical note about an English-specific subset of Unicode (technically called 'Basic Latin' in Unicode, and commonly referred to as ASCII). Starting in the 1960s, the Latin characters used to write English (like the ones used in this sentence), were represented with a code called **ASCII** (American Standard Code for Information Interchange). ASCII represented each character with a single byte. A byte can represent 256 different characters, but ASCII only used 127 of them; the high-order bit of ASCII bytes is always set to 0. (Actually it only used 95 of them and the rest were control codes for an obsolete machine called a teletype). Here's a few ASCII characters with their representation in hex and decimal:

| Ch | Hex | Dec | Ch | Hex | Dec | Ch | Hex | Dec | Ch | Hex | Dec | |
|----|-----|-----|----|-----|-----|-----|-----|-----|----|-----|-----|----|
| < | 3C | 60 | @ | 40 | 64 | ... | \ | 5C | 92 | ` | 60 | 96 |
| = | 3D | 61 | A | 41 | 65 | ... | [| 5D | 93 | a | 61 | 97 |
| > | 3E | 62 | B | 42 | 66 | ... | ^ | 5E | 94 | b | 62 | 98 |
| ? | 3F | 63 | C | 43 | 67 | ... | _ | 5F | 95 | c | 63 | 99 |

Figure 2.4 Some selected ASCII codes for some English letters, with the codes shown both in hexadecimal and decimal.

But ASCII is of course insufficient since there are lots of other characters in the world's writing systems! Even for scripts that use Latin characters, there are many more than the 95 in ASCII. For example, this Spanish phrase (meaning "Sir, replied Sancho") has two non-ASCII characters, ñ and ó:

(2.10) Señor- respondió Sancho-

Devanagari And lots of languages aren't based on Latin characters at all! The **Devanagari** script is used for 120 languages (including Hindi, Marathi, Nepali, Sindhi, and Sanskrit). Here's a Devanagari example from the Hindi text of the Universal Declaration of Human Rights:

अनुच्छेद १(एक): सभी मनुष्य जन्म से स्वतन्त्र तथा मर्यादा और अधिकारों में समान होते हैं। वे तर्क और विवेक से सम्पन्न हैं तथा उन्हें भ्रातृत्व की भावना से परस्पर के प्रति कार्य करना चाहिए।

Chinese has about 100,000 Chinese characters in Unicode (including overlapping and non-overlapping variants used in Chinese, Japanese, Korean, and Vietnamese, collectively referred to as CJKV).

All in all there are more than 150,000 characters and 168 different scripts supported in Unicode 16.0. Even though many scripts from around the world have yet to be added to Unicode, there are so many there, from scripts used by modern languages (Chinese, Arabic, Hindi, Cherokee, Ethiopic, Khmer, N'Ko, Turkish,

Spanish) to scripts of ancient languages (Cuneiform, Ugaritic, Egyptian Hieroglyph, Pahlavi), as well as mathematical symbols, emojis, currency symbols, and more.

2.3.1 Code Points

code point How does it work? Unicode assigns a unique id, called a **code point**, for each one of these 150,000 characters.

The code point is an abstract representation of the character, and each code point is represented by a number, traditionally written in hexadecimal, from number 0 through 0x10FFFF (which is 1,114,111 decimal). Having over a million code points means there is a lot of room for new characters. It is traditional to represent these code points with the prefix “U+” (which just means “the following is a Unicode hex representation of a code point”). So the code point for the character a is U+0061 which is the same as 0x0061. (Note that Unicode was designed to be backwards compatible with ASCII, which means that the first 127 code points, including the code for a, are identical with ASCII.) Here are some sample code points; some (but not all) come with descriptions:

| | | |
|---------|----|--------------------------------------|
| U+0061 | a | LATIN SMALL LETTER A |
| U+0062 | b | LATIN SMALL LETTER B |
| U+0063 | c | LATIN SMALL LETTER C |
| U+00F9 | ù | LATIN SMALL LETTER U WITH GRAVE |
| U+00FA | ú | LATIN SMALL LETTER U WITH ACUTE |
| U+00FB | û | LATIN SMALL LETTER U WITH CIRCUMFLEX |
| U+00FC | ü | LATIN SMALL LETTER U WITH DIAERESIS |
| U+8FDB | 进 | |
| U+8FDC | 远 | |
| U+8FDD | 违 | |
| U+8FDE | 连 | |
| U+1F600 | 😊 | GRINNING FACE |
| U+1F00E | 入箭 | MAHJONG TILE EIGHT OF CHARACTERS |

glyph Note that a code point does not specify the **glyph**, the visual representation of a character. Glyphs are stored in **fonts**. The code point U+0061 is an abstract representation of a. There can be an indefinite number of visual representations, for example in different fonts like Times Roman (a) or Courier (a), or different font styles like boldface (**a**) or italic (*a*). But all of them are represented by the same code point U+0061.

2.3.2 UTF-8 Encoding

While the code point (the unique id) is the abstract Unicode representation of the character, we don’t just stick that id in a text file.

encoding Instead, whenever we need to represent a character in a text string, we write an **encoding** of the character. There are many different possible encoding methods, but the encoding method called UTF-8 is by far the most frequent (for example almost the entire web is encoded in UTF-8).

Let’s talk about encodings. The Unicode representation of the word `hello` consists of the following sequence of 5 code points:

U+0068 U+0065 U+006C U+006C U+006F

We can imagine a very simple encoding method: just write the code point id in a file. Since there are more than 1 million characters, 16 bits (2 bytes) isn't enough, so we'll need to use 4 bytes (32 bit) to capture the 21 bits we need to represent 1.1 million characters. (We could fit it in 3 bytes but it's inconvenient to use multiples of 3 for bytes.)

With this 4-byte representation the word `hello` would be encoded as the following set of bytes:

```
00 00 00 68 00 00 00 65 00 00 00 6C 00 00 00 6C 00 00 00 6F
```

But we don't use this encoding (which is technically called UTF-32) because it makes every file 4 times longer than it would have been in ASCII, making files really big and full of zeros. Also those zeros cause another problem: it turns out that having any byte that is completely zero messes things up for backwards compatibility for ASCII-based systems that historically used a 0 byte as an end-of-string marker.

UTF-8
variable-length
encoding

Instead, the most common encoding standard is **UTF-8** (Unicode Transformation Format 8), which represents characters efficiently (using fewer bytes on average) by writing some characters using fewer bytes and some using more bytes. UTF-8 is thus a **variable-length encoding**.

For some characters (the first 127 code points, i.e. the set of ASCII characters), UTF-8 encodes them as a single byte, so the UTF-8 encoding of `hello` is :

```
68 65 6C 6C 6F
```

This conveniently means that files encoded in ASCII are also valid UTF-8 encodings!

But UTF-8 is a variable length encoding, meaning that code points ≥ 128 are encoded as a sequence of two, three, or four bytes. Each of these bytes are between 128 and 255, so they won't be confused with ASCII, and each byte indicates in the first few bits whether it's a 2-byte, 3-byte, or 4-byte encoding.

| From - To | Code Points | UTF-8 Encoding | | | |
|-------------------|--------------------------|----------------|---------|---------|---------|
| | Bit Value | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
| U+0000-U+007F | 0xxxxxxx | xxxxxxx | | | |
| U+0080-U+07FF | 0000yyy yyxxxxx | 110yyyy | 10xxxxx | | |
| U+0800-U+FFFF | zzzyyyy yyxxxxx | 1110zzz | 10yyyyy | 10xxxxx | |
| U+010000-U+10FFFF | 000uuuuu zzzyyyy yyxxxxx | 11110uuu | 10uuzzz | 10yyyyy | 10xxxxx |

Figure 2.5 Mapping from Unicode code point to the variable length UTF-8 encoding. For a given code point in the From-To range, the bit value in column 2 is packed into 1, 2, 3, or 4 bytes. Figure adapted from Unicode 16.0 Core Spec Chapter 3 Table 3-6.

Fig. 2.5 shows how this mapping occurs. For example these rules explain how the character ñ, which has code point U+00F1, or bit sequence `00000000 11110001`, (where blue indicates the sequence `yyyyy` and red the sequence `xxxxxx`) is encoded into to the two-byte bit sequence `11000011 10110001` or `0xC3B1`. As a result of these rules, the first 127 characters (ASCII) are mapped to one byte, most remaining characters in European, Middle Eastern, and African scripts map to two bytes, most Chinese, Japanese, and Korean characters map to three bytes, and rarer CJKV characters and emojis and some symbols map to 4 bytes.

UTF-8 has a number of advantages. It's relatively efficient, using fewer bytes for commonly-encountered characters, it doesn't use zero bytes (except when literally representing the NULL character which is U+0000), it's backwards compatible with ASCII, and it's self-synchronizing, meaning that if a file is corrupted, it's always possible to find the start of the next or prior character just by moving up to 3 bytes left or right.

Unicode and Python: Starting with Python 3, all Python strings are stored internally as Unicode, each string a sequence of Unicode code points. Thus string functions and regular expressions all apply natively to code points. For example, functions like `len()` of a string return its length in characters, i.e., code points, not its length in bytes.

When reading or writing from a file, however, the code points need to be encoded and decoding using a method like UTF-8. That is, every file is encoded in some encoding. If it's not UTF-8, it's an older encoding method like ASCII or Latin-1 (`iso_8859_1`). There is no such thing as a text file without an encoding. The encoding method is specified in Python when opening a file for reading and writing.

2.4 Subword Tokenization: Byte-Pair Encoding

tokenization
tokens

Tokenization, the first stage of natural language processing, is the process of segmenting the running input text into **tokens**.

We've seen three candidates for tokens: words, morphemes and characters. But each has problems as a unit. Words and morphemes seem approximately at the right level for NLP processing, since they tend to have consistent meanings, but they are challenging to define formally. Characters are clearer to define, but seem too small a unit to choose for tokens.

In this section we introduce what we do in practice for NLP: use a data-driven approach to define tokens that will generally result in units about the size of morphemes or words, but occasionally use units as small as characters.

Why tokenize the input? One reason is that converting an input to a deterministic fixed set of units means that different algorithms and systems can agree on simple questions. For example, How long is this text? (How many units are in it?). Or: Is *don't* or *New York* one token or two? Standardizing is thus essential for replicability in NLP experiments, and many algorithms that we introduce in this book (like the **perplexity** metric for language models) assume that all texts have a fixed tokenization.

Tokenization algorithms that include smaller tokens for morphemes and letters also eliminate the problem of **unknown words**. What are these? As we will see in the next chapter, NLP algorithms often learn some facts about language from one corpus (a **training** corpus) and then use these facts to make decisions about a separate **test** corpus and its language. Thus if our training corpus contains, say the words *low*, *new*, and *newer*, but not *lower*, then if the word *lower* appears in our test corpus, our system will not know what to do with it.

subwords

To deal with this unknown word problem, modern tokenizers automatically induce sets of tokens that include tokens smaller than words, called **subwords**. Subwords can be arbitrary substrings, or they can be meaning-bearing units like the morphemes *-est* or *-er*. In modern tokenization schemes, many tokens are words, but other tokens are frequently occurring morphemes or other subwords like *-er*. Every unseen word can thus be represented by some sequence of known subword units. For example, if we had happened not to ever see the word *lower*, when it appears we could segment it successfully into *low* and *er* which we had already seen. In the worst case, a really unusual word (perhaps an acronym like *GRPO*) could be tokenized as a sequence of individual letters if necessary.

Two tokenization algorithms are widely used in modern language models: **byte-pair encoding** (BPE) (Sennrich et al., 2016), and **unigram language modeling**

BPE (ULM) (Kudo, 2018).² In this section we introduce the **byte-pair encoding** or **BPE** algorithm (Sennrich et al., 2016; Gage, 1994); see Fig. 2.6.

Like most tokenization schemes, the BPE algorithm has two parts: a **trainer**, and an **encoder**. In general in the token training phase we take a raw training corpus (usually roughly pre-separated into words, for example by whitespace) and induce a vocabulary, a set of tokens. Then a token encoder takes a raw test sentence and encodes it into the tokens in the vocabulary that were learned in training.

2.4.1 BPE training

The **BPE** training algorithm iteratively merges frequent neighboring tokens to create longer and longer tokens. The algorithm begins with a vocabulary that is just the set of all individual characters. It then examines the training corpus, and finds the two characters that are most frequently adjacent. Imagine our original corpus is 10 characters long, using a vocabulary of 5 characters, {A, B, C, D, E}:

A B D C A B E C A B

The most frequent neighboring pair of characters is “A B” so we merge those, add a new merged token ‘AB’ to the vocabulary, and replace every adjacent ‘A’ ‘B’ in the corpus with the new ‘AB’:

AB D C AB E C AB

Now we have a vocabulary of 6 possible tokens {A, B, C, D, E, AB}, and the corpus has length 7. And now the most frequent pair of tokens is “C AB”, so we merge those, leading to a vocabulary with 7 tokens {A, B, C, D, E, AB, CAB}, and the corpus has length 5.

AB D CAB E CAB

The algorithm continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus k new symbols. That’s the core of the algorithm.

The only additional complication is that in practice, instead of running on the raw sequence of characters, the algorithm is usually run only *inside* words. That is, the algorithm does not merge across word boundaries. To do this, the input corpus is often first separated at white space and punctuation (using the regular expressions that we define later in the chapter). This gives a starting set of strings, each corresponding to the characters of a word, (with the white space usually attached to the start of the word), together with the counts of the words. Then while counts come from a corpus, merges are only allowed within the strings.

Let’s see how the full algorithm thus works on this tiny synthetic corpus, where we’ve explicitly marked the spaces between words:³

(2.11) `set_new_new_renew_reset_renew`

First, we’ll break up the corpus into words, with leading whitespace, together with their counts; no merges will be allowed to go beyond these word boundaries. The result looks like the following list of 4 words and a starting vocabulary of 7 characters:

² The **SentencePiece** library includes implementations of both of these (Kudo and Richardson, 2018a), and people sometimes use the name **SentencePiece** to simply mean ULM tokenization.

³ Yes, we realize this isn’t a particularly likely or exciting sentence.

| corpus | vocabulary |
|------------------|---------------------|
| 2 ↵ n e w | ↵, e, n, r, s, t, w |
| 2 ↵ r e n e w | |
| 1 s e t | |
| 1 ↵ r e s e t | |

The BPE training algorithm first counts all pairs of adjacent symbols: the most frequent is the pair `n e` because it occurs in `new` (frequency of 2) and `renew` (frequency of 2) for a total of 4 occurrences. We then merge these symbols, treating `ne` as one symbol, and count again:

| corpus | vocabulary |
|------------------|-------------------------|
| 2 ↵ n e w | ↵, e, n, r, s, t, w, ne |
| 2 ↵ r e n e w | |
| 1 s e t | |
| 1 ↵ r e s e t | |

Now the most frequent pair is `ne w` (total count=4), which we merge.

| corpus | vocabulary |
|------------------|------------------------------|
| 2 ↵ n e w | ↵, e, n, r, s, t, w, ne, new |
| 2 ↵ r e n e w | |
| 1 s e t | |
| 1 ↵ r e s e t | |

Next `↵ r` (total count of 3) get merged to `↵r`, and then `↵r e` (total count 3) gets merged to `↵re`. The system has essentially induced that there is a word-initial prefix `re-`:

| corpus | vocabulary |
|----------------|---------------------------------------|
| 2 ↵ n e w | ↵, e, n, r, s, t, w, ne, new, ↵r, ↵re |
| 2 ↵re n e w | |
| 1 s e t | |
| 1 ↵re s e t | |

If we continue, the next merges are:

| merge | current vocabulary |
|------------|--|
| (↵, new) | ↵, e, n, r, s, t, w, ne, new, ↵r, ↵re, ↵new |
| (↵re, new) | ↵, e, n, r, s, t, w, ne, new, ↵r, ↵re, ↵new, ↵renew |
| (s, e) | ↵, e, n, r, s, t, w, ne, new, ↵r, ↵re, ↵new, ↵renew, se |
| (se, t) | ↵, e, n, r, s, t, w, ne, new, ↵r, ↵re, ↵new, ↵renew, se, set |

function BYTE-PAIR ENCODING(strings C , number of merges k) **returns** vocab V

```

 $V \leftarrow$  all unique characters in  $C$            # initial set of tokens is characters
for  $i = 1$  to  $k$  do                           # merge tokens  $k$  times
     $t_L, t_R \leftarrow$  Most frequent pair of adjacent tokens in  $C$ 
     $t_{NEW} \leftarrow t_L + t_R$                  # make new token by concatenating
     $V \leftarrow V + t_{NEW}$                        # update the vocabulary
    Replace each occurrence of  $t_L, t_R$  in  $C$  with  $t_{NEW}$  # and update the corpus
return  $V$ 

```

Figure 2.6 The training part of the BPE algorithm for taking a corpus broken up into individual characters or bytes, and learning a vocabulary by iteratively merging tokens. Figure adapted from [Bostrom and Durrett \(2020\)](#).

2.4.2 BPE encoder

Once we've learned our vocabulary, the BPE **encoder** is used to tokenize a test sentence. The encoder just runs on the test data the merges we have learned from the training data. It runs them greedily, in the order we learned them. (Thus the frequencies in the test data don't play a role, just the frequencies in the training data). So first we segment each test sentence word into characters. Then we apply the first rule: replace every instance of `n e` in the test corpus with `ne`, and then the second rule: replace every instance of `ne w` in the test corpus with `new`, and so on. By the end of course many of the merges simply recreated words in the training set. But the merges also created knowledge of morphemes like the `re-` prefix (that might appear in perhaps unseen combinations like `revisit` or `rearrange`), or the morpheme `new` without an initial space (hence word-internal) that might appear at the start of sentences or in words unseen in training like `anew`.

Of course in real settings BPE is run with tens of thousands of merges on a very large input corpus, to produce vocabulary sizes of 50,000, 100,000, or even 200,000 tokens. The result is that most words can be represented as single tokens, and only the rarer words (and unknown words) will have to be represented by multiple tokens. At least for English. For multilingual systems, the tokens can be dominated by English, leaving fewer tokens for other languages, as we'll discuss below.

2.4.3 BPE in practice

The example above just showed simple BPE learning from sequences of ASCII bytes. How does BPE work with Unicode input? We normally run BPE on the individual bytes of UTF-8-encoded text. That is, we take a Unicode representations of text as a series of code points, encode it in bytes using UTF-8, and we treat each of these individual bytes as the input to BPE. Thus BPE likely begins by rediscovering the 2-byte and common 3-byte sequences that UTF-8 uses to encode various code points. Again, running BPE only inside presegmented words helps avoid problems. Because there are only 256 possible values of a byte, there will be no unknown tokens, although it's possible that BPE will learn some illegal UTF-8 sequences across character boundaries. These will be very rare, and can be eliminated with a filter.

Let's see some examples of the industrial application of the BPE tokenizer used in large systems like OpenAI GPT4o. This tokenizer has 200K tokens, which is a comparatively large number. We can use Tat Dat Duong's Tiktokenizer visualizer (<https://tiktokenizer.vercel.app/>) to see the number of tokens in a given sentence. For example here's the tokenization of a nonsense sentence we made up; the visualizer uses a center dot to indicate a space:

Anyhow, · she 's · seen · Jane 's · 224123 · flowers · anyhow!

The visualization shows colors to separate out words, but of course the true output of the tokenizer is simply a sequence of unique token ids. (In case you're interested, they were the following 13 tokens: 11865, 8923, 11, 31211, 6177, 23919, 885, 220, 19427, 7633, 18887, 147065, 0)

Notice that most words are their own token, usually including the leading space. Clitics like `'s` are segmented off when they appear on proper nouns like `Jane`, but are counted as part of a word for frequent words like `she's`. Numbers tend to be segmented into chunks of 3 digits. And some words (like `anyhow`) are segmented differently if they appear capitalized sentence-initially (two tokens, `Any` and `how`), then if they appear after a space, lower case (one token `anyhow`).

pretokenization Some of these are related to preprocessing steps. As we mentioned briefly above, language models usually create their tokens in a **pretokenization** stage that first segments the input using regular expressions, for example breaking the input at spaces and punctuation, stripping off clitics, and breaking numbers into sets of digits. We'll see how to use regular expressions in Section 2.6.

SuperBPE
BoundlessBPE It's possible to change this pretokenization to allow BPE tokens to span multiple words. For example the **SuperBPE** (Liu et al., 2025) and **BoundlessBPE** (Schmidt et al., 2025) algorithms first induce regular BPE subword tokens by enforcing pretokenization. They then run a second stage of BPE allowing merges across spaces and punctuation. The result is a large set of tokens that can be more efficient (Fig. 2.7).

| | |
|-----------|--|
| BPE: | By the way, I am a fan of the Milky Way. |
| SuperBPE: | By the way, I am a fan of the Milky Way. |

Figure 2.7 The SuperBPE algorithm creating larger tokens by allowing a second stage of merging across spaces. Figure from Liu et al. (2025).

Many of the tokenizers used in practice for large language models are multilingual, trained on many languages. But because the training data for large language models is vastly dominated by English text, these multilingual BPE tokenizers tend to use most of the tokens for English, leaving fewer of them for other languages. The result is that they do a better job of tokenizing English, and the other languages tend to get their words split up into shorter tokens. For example let's look at a Spanish sentence from a recipe for plantains, together with an English translation.

The English has 18 tokens; each of the 14 words is a token (none of the words are split into multiple tokens):

In·a·deep·bowl,·mix·the·orange·juice·with·the·sugar,·ginger,·and·nutmeg.

By contrast, the original 16 words in Spanish have been encoded into 33 tokens, a much larger number. Notice that many basic words have been broken into pieces. For example *hondo*, 'deep', has been segmented into *h* and *ondo*. Similarly for *jugo*, 'juice', *nuez*, 'nut' and *jengibre* 'ginger':

En·un·recipiente·hondo,·mezclar·el·jugo·de·naranja·con·el·azúcar,·jengibre,·y·nuez·moscada.

Spanish is not a particularly low-resource language; this oversegmenting can be even more serious in lower resource languages, often down to individual characters. Oversegmenting into these tiny tokens can cause various problems for the downstream processing of the language. As will become more clear once we introduce transformer models in Chapter 8, such fragmentation can lead to poor representations of meaning, the need for longer contexts, and higher costs to train models (Rust et al., 2021; Ahia et al., 2023).

2.5 Corpora

Words don't appear out of nowhere. Any particular piece of text that we study is produced by one or more specific speakers or writers, in a specific dialect of a

specific language, at a specific time, in a specific place, for a specific function.

Perhaps the most important dimension of variation is the language. NLP algorithms are most useful when they apply across many languages. The world has 7097 languages at the time of this writing, according to the online Ethnologue catalog (Simons and Fennig, 2018). It is important to test algorithms on more than one language, and particularly on languages with different properties; by contrast there is an unfortunate current tendency for NLP algorithms to be developed or tested just on English (Bender, 2019). Even when algorithms are developed beyond English, they tend to be developed for the official languages of large industrialized nations (Chinese, Spanish, Japanese, German etc.), but we don't want to limit tools to just these few languages. Furthermore, most languages also have multiple varieties, often spoken in different regions or by different social groups. Thus, for example, if we're processing text that uses features of African American English (AAE) or African American Vernacular English (AAVE)—the variations of English that can be used by millions of people in African American communities (King 2020)—we must use NLP tools that function with features of those varieties. Twitter posts might use features often used by speakers of African American English, such as constructions like *iont* (*I don't* in Mainstream American English (MAE)), or *talmbout* corresponding to MAE *talking about*, both examples that influence word segmentation (Blodgett et al. 2016, Jones 2015).

AAE

MAE

code switching

It's also quite common for speakers or writers to use multiple languages in a single utterance, a phenomenon called **code switching**. Code switching is enormously common across the world; here are examples showing Spanish and (transliterated) Hindi code switching with English (Solorio et al. 2014, Jurgens et al. 2017):

(2.12) Por primera vez veo a @username actually being hateful! it was beautiful:
 [*For the first time I get to see @username actually being hateful! it was beautiful:*]]

(2.13) dost tha or ra- hega ... dont worry ... but dherya rakhe
 [*“he was and will remain a friend ... don't worry ... but have faith”*]

Another dimension of variation is the genre. The text that our algorithms must process might come from newswire, fiction or non-fiction books, scientific articles, Wikipedia, or religious texts. It might come from spoken genres like telephone conversations, business meetings, police body-worn cameras, medical interviews, or transcripts of television shows or movies. It might come from work situations like doctors' notes, legal text, or parliamentary or congressional proceedings.

Text also reflects the demographic characteristics of the writer (or speaker): their age, gender, race, socioeconomic class can all influence the linguistic properties of the text we are processing.

And finally, time matters too. Language changes over time, and for some languages we have good corpora of texts from different historical periods.

Because language is so situated, when developing computational models for language processing from a corpus, it's important to consider who produced the language, in what context, for what purpose. How can a user of a dataset know all these details? The best way is for the corpus creator to build a **datasheet** (Geburu et al., 2020) or **data statement** (Bender et al., 2021) for each corpus. A datasheet specifies properties of a dataset like:

datasheet

Motivation: Why was the corpus collected, by whom, and who funded it?

Situation: When and in what situation was the text written/spoken? For example, was there a task? Was the language originally spoken conversation, edited text, social media communication, monologue vs. dialogue?

- Language variety:** What language (including dialect/region) was the corpus in?
- Speaker demographics:** What was, e.g., the age or gender of the text’s authors?
- Collection process:** How big is the data? If it is a subsample how was it sampled? Was the data collected with consent? How was the data pre-processed, and what metadata is available?
- Annotation process:** What are the annotations, what are the demographics of the annotators, how were they trained, how was the data annotated?
- Distribution:** Are there copyright or other intellectual property restrictions?

2.6 Regular Expressions

regular
expression

One of the most useful tools for text processing in computer science is the **regular expression** (or **regex**), a language for specifying text strings. Regexes are used in every computer language, in text processing tools like Unix `grep`, and in editors like vim or Emacs. And they play an important role in the pre-tokenization step for tokenization algorithms like BPE. Formally, a regular expression is an algebraic notation for characterizing a set of strings. Practically, we can use a regex to search for a string in a text and to specify how to change the string, both of which are key to tokenization.

string

We use regular expressions to search for a **pattern** in a **string** which can be a single line or a longer text. For example, the Python function

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

scans through the **string** and returns the first match inside it for the **pattern**. In the following examples we generally highlight the exact string that matches the regular expression and show only the first match. We’ll use Python syntax, expressing the regex as a raw string delimited by double quotes: `r"regex"`. Raw strings treat backslashes as literal characters, which will be important since many regex patterns we’ll introduce use backslashes.

Regular expressions come in different variants, so using an online regex tester can help make sure your regex does what you think it’s doing.

2.6.1 Character Disjunction: The Square Bracket

character
disjunction

The simplest kind of regular expression is a sequence of simple characters. The pattern `r"Buttercup"` matches the substring `Buttercup` in any string (like the string `I’m called little Buttercup`). But often we need to use special characters. For example, we might want to match *either* some character or another. For example, regular expressions are generally **case sensitive**: `r"s"` matches a lower case `s` but not an upper case `S`. To match both `s` and `S` we can use the **character disjunction** operator, the square braces `[` and `]`. The string of characters inside the braces specifies a disjunction of characters to match. For example, Fig. 2.8 shows that the pattern `r"[mM]"` matches patterns containing either `m` or `M`.

| Pattern | Match | String |
|------------------------------|------------------|---------------------------------------|
| <code>r"[mM]ary"</code> | Mary or mary | “ <u>M</u> ary Ann stopped by Mona’s” |
| <code>r"[abc]"</code> | ‘a’, ‘b’, or ‘c’ | “In uomini, in soldati” |
| <code>r"[1234567890]"</code> | any one digit | “plenty of <u>7</u> to 5” |

Figure 2.8 The use of the brackets `[]` to specify a disjunction of characters.

The regular expression `r"[1234567890]"` specifies any single digit. This can get awkward (imagine typing `r"[ABCDEFGHJKLMNOPQRSTUVWXYZ]"` to mean an uppercase letter) so the brackets can also be used with a dash (-) to specify any one character in a **range**. The pattern `r"[2-5]"` specifies any one of the characters 2, 3, 4, or 5. The pattern `r"[b-g]"` specifies one of the characters *b*, *c*, *d*, *e*, *f*, or *g*. Some other examples are shown in Fig. 2.9.

| Regex | Match | Example Patterns Matched |
|-----------------------|----------------------|---|
| <code>r"[A-Z]"</code> | an upper case letter | "we should call it 'Drenched Blossoms'" |
| <code>r"[a-z]"</code> | a lower case letter | "my beans were impatient to be hoed!" |
| <code>r"[0-9]"</code> | a single digit | "Chapter 1: Down the Rabbit Hole" |

Figure 2.9 The use of the brackets [] plus the dash - to specify a range.

The square braces can also be used to specify what a single character *cannot* be, by use of the caret ^. If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated. For example, the pattern `r"[^a]"` matches any single character (including special characters) except *a*. This is only true when the caret is the first symbol after the open square brace. If it occurs anywhere else, it usually stands for a caret; Fig. 2.10 shows some examples.

| Regex | Match (single characters) | Example Patterns Matched |
|------------------------|---------------------------|------------------------------------|
| <code>r"[^A-Z]"</code> | not an upper case letter | "Oyfn pripetchik" |
| <code>r"[^Ss]"</code> | neither 'S' nor 's' | "I have no exquisite reason for't" |
| <code>r"[^.]"</code> | not a period | "our resident Djinn" |
| <code>r"[e^]"</code> | either 'e' or '^' | "look up ^ now" |
| <code>r"a^b"</code> | the pattern 'a^b' | "look up a^ b now" |

Figure 2.10 The caret ^ for negation or just to mean ^. See below re: the backslash for escaping the period.

2.6.2 Counting, Optionality, and Wildcards

How can we talk about optional elements, like an optional *s* if we want to match both *koala* and *koalas*? We can't use the square brackets, because while they allow us to say "s or S", they don't allow us to say "s or nothing". For this we use the question mark `r"?"`, which means "the preceding character or nothing". So `r"colou?r"` matches both *color* and *colour*, and `r"koalas?"` matches *koala* or *koalas*.

There's another way to talk about elements that may or may not occur. Consider the language of certain sheep, which consists of strings that look like the following:

```
baa!
baaa!
baaaa!
...
```

This sheep language consists of strings with a *b*, followed by at least two (and arbitrarily more) *a*'s, followed by an exclamation point. To represent this language, we'll use a useful operator that is represented by the asterisk or `*`, called the **Kleene *** (generally pronounced "cleany star"). The Kleene star means "zero or more occurrences of the immediately previous character or regular expression". So `r"a^b"` means "any string of zero or more as".

Kleene *

Could `r"ba^b"` represent the sheep language? It will correctly match *ba* or *baaaaa*, but there's a problem! It will also match *b*, with no *a*, or *ba* with only one

a. That’s because Kleene star means “zero or more occurrences”. Instead, for the sheep language we’ll want `r"baaa*"`, meaning `b` followed by `aa` followed by zero or more additional `as`. More complex patterns can also be repeated. So `r"[ab]*"` means “zero or more `a`’s or `b`’s” (not “zero or more right square braces”). This will match strings like `aaaa` or `ababab` or `bbbb`, as well as the empty string. For specifying an integer (a string of digits) we can use `r"[0-9][0-9]*"`. (Why isn’t it just `r"[0-9]*"`?)

Kleene + There is a slightly shorter way to specify “at least one” of some character: the **Kleene +**, which means “one or more occurrences of the immediately preceding character or regular expression”. So `r"[0-9]+"` is the normal way to specify “a sequence of digits”, and we could also specify the sheep language as `r"baa+!"`.

Besides the Kleene `*` and Kleene `+` we can also use explicit numbers as counters, by enclosing them in curly brackets. The operator `r"{3}"` means “exactly 3 occurrences of the previous character or expression”. So `r"ax{10}z"` will match a followed by exactly 10 `x`’s followed by `z`.

period An important special character is the **period** (`r"."`), a **wildcard** expression that matches any single character (*except* a newline).

The wildcard is often used together with the Kleene star to mean “any string of characters”. For example, suppose we want to find any line in which a particular word, for example, `rose`, appears twice. We can specify this with the regular expression `r"rose.*rose"`, meaning two `roses`, with a sequence of zero or more characters (of any kind) between them. Fig. 2.11 summarizes.

| Regex | Match |
|------------------|---|
| <code>*</code> | zero or more occurrences of the previous char or expression |
| <code>+</code> | one or more occurrences of the previous char or expression |
| <code>?</code> | zero or one occurrence of the previous char or expression |
| <code>{n}</code> | exactly n occurrences of the previous char or expression |
| <code>.</code> | any single char |
| <code>.*</code> | any string of zero or more chars |

Figure 2.11 Counting and wildcards.

2.6.3 Anchors and Boundaries

anchors **Anchors** are special characters that anchor regular expressions to particular places in a string. The most common anchors are the caret `^` and the dollar sign `$`. The caret `^` matches the start of a line. The pattern `r"^The"` matches the word `The` only at the start of a line. Thus, the caret `^` has three uses: to match the start of a line, to indicate a negation inside of square brackets, and just to mean a caret. (What are the contexts that allow the system to know which function a given caret is supposed to have?) The dollar sign `$` matches the end of a line. So the pattern `␣$` is a useful pattern for matching a space at the end of a line, and `r"^The dog\.$"` matches a line that contains only the phrase `The dog.` with a final period.

Note that we have to use the backslash in the prior example since we want the `.` to mean “period” and not the wildcard. By contrast, the regular expression `r"^The dog.$"` would match `The dog.` but also `The dog!` and `The dogo.` As we’ll discuss below, all the special characters we’ve defined so far (`*` `+` `?` `.` `[` `]`) need to be backslashed when we mean to use them literally.

There are other anchors: `\b` matches a word boundary, and `\B` matches a non word-boundary. Thus, `r"\bthe\b"` matches the word `the` but not the word `other`.

| Regex | Match |
|-----------------|-------------------|
| <code>^</code> | start of line |
| <code>\$</code> | end of line |
| <code>\b</code> | word boundary |
| <code>\B</code> | non-word boundary |

Figure 2.12 Anchors in regular expressions.

A “word” for the purposes of a regex is defined (based on words in programming languages) as a sequence of digits, underscores, or letters. Thus `r"\b99\b"` will match the string `99` in `There are 99 bottles of beer on the wall` (because `99` follows a space) but not `99` in `There are 299 bottles of beer on the wall` (since `99` follows a number). But it will match `99` in `$99` (since `99` follows a dollar sign (`$`), which is not a digit, underscore, or letter).

Note that all these anchors and boundary operators technically match the empty string, meaning that they don’t eat up any characters of the string. The carat in the pattern `r"^The"` matches the start of `"The"` but doesn’t actually advance over the first character `T`. And the pattern `r"the\b the"` matches `the the`; the `\b` is aware of the fact that the space is a boundary, but it matches the empty string right before the space, not the space, so that the space character is available to be matched.

2.6.4 Disjunction, Grouping, and Precedence

Suppose we need to search for texts about pets; perhaps we are particularly interested in cats and dogs. In such a case, we might want to search for either the string `cat` or the string `dog`. Since we can’t use the square brackets to search for “cat or dog” (why wouldn’t `r"[catdog]"` do the right thing?), we need a new operator, **disjunction** the **disjunction** operator, also called the **pipe** symbol `|`. The pattern `r"cat|dog"` matches either the string `cat` or the string `dog`.

Sometimes we need to use this disjunction operator in the midst of a larger sequence. For example, suppose I want to search for mentions of pet fish. How can I specify both `guppy` and `guppies`? We cannot simply say `r"guppy|ies"`, because that would match only the strings `guppy` and `ies`. This is because sequences like `guppy` take **precedence** over the disjunction operator `|`. To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators `(` and `)`. Enclosing a pattern in parentheses makes it act like a single character for the purposes of neighboring operators like the pipe `|` and the Kleene*. So the pattern `r"gupp(y|ies)"` would specify that we meant the disjunction only to apply to the suffixes `y` and `ies`.

The parenthesis operator `(` is also useful when we are using counters like the Kleene*. Unlike the `|` operator, the Kleene* operator applies by default only to a single character, not to a whole sequence. Suppose we want to match repeated instances of a string. Perhaps we have a line that has column labels of the form `Column 1 Column 2 Column 3`. The expression `r"Column_[0-9]+_*` will not match any number of columns; instead, it will match a single column followed by any number of spaces! The star here applies only to the space `_` that precedes it, not to the whole sequence. With the parentheses, we could write the expression `r"(Column_[0-9]+_+)*"` to match the word `Column`, followed by a number and spaces, the whole pattern repeated zero or more times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the **operator precedence hierarchy** for regular expressions. The following table gives the order

of operator precedence, from highest precedence to lowest precedence.

| | |
|-----------------------|---------------|
| Parenthesis | () |
| Counters | * + ? {} |
| Sequences and anchors | the ^my end\$ |
| Disjunction | |

Thus, because counters have a higher precedence than sequences, `r"the*"` matches *theeee* but not *thethe*. Because sequences have a higher precedence than disjunction, `r"the|any"` matches *the* or *any* but not *thany* or *they*.

Patterns can be ambiguous in another way. Consider the expression `r"[a-z]*"` when matching against the text *once upon a time*. Since `r"[a-z]*"` matches zero or more letters, this expression could match nothing, or just the first letter *o*, *on*, *onc*, or *once*. In these cases regular expressions always match the *largest* string they can; we say that patterns are **greedy**, expanding to cover as much of a string as they can.

greedy
non-greedy

There are, however, ways to enforce **non-greedy** matching, using another meaning of the `?` qualifier. The operator `*?` is a Kleene star that matches as little text as possible. The operator `+?` is a Kleene plus that matches as little text as possible.

*?
+?

2.6.5 A Simple Example

Suppose we wanted to write a regex to find cases of the English article *the*. A simple (but incorrect) pattern might be:

```
r"the" (2.14)
```

One problem is that this pattern will miss the word when it begins a sentence and hence is capitalized (i.e., *The*). This might lead us to the following pattern:

```
r"[tT]he" (2.15)
```

But we will still overgeneralize, incorrectly return texts with `the` embedded in other words (e.g., *other* or *there*). So we need to specify that we want instances with a word boundary on both sides:

```
r"\b[tT]he\b" (2.16)
```

The simple process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like *other* or *there*, and **false negatives**, strings that we incorrectly missed, like *The*. Addressing these two kinds of errors comes up again and again in language processing. Reducing the overall error rate for an application thus involves two antagonistic efforts:

false positives
false negatives

- Increasing **precision** (minimizing false positives)
- Increasing **recall** (minimizing false negatives)

We'll come back to precision and recall with more precise definitions in Chapter 4.

2.6.6 More Operators

Figure 2.13 shows some useful aliases for common ranges:

newline

Finally, certain special characters are referred to by special notation based on the backslash (`\`) (see Fig. 2.14). The most common of these are the **newline** character `\n` and the **tab** character `\t`.

| Regex | Expansion | Match | First Matches |
|-----------------|---------------------------|-----------------------------|---------------|
| <code>\d</code> | <code>[0-9]</code> | any digit | Party_of_5 |
| <code>\D</code> | <code>[^0-9]</code> | any non-digit | Blue_moon |
| <code>\w</code> | <code>[a-zA-Z0-9_]</code> | any alphanumeric/underscore | Daiyu |
| <code>\W</code> | <code>[^\w]</code> | a non-alphanumeric | !!!! |
| <code>\s</code> | <code>[\r\t\n\f]</code> | whitespace (space, tab) | in_Concord |
| <code>\S</code> | <code>[^\s]</code> | Non-whitespace | in_Concord |

Figure 2.13 Aliases for common sets of characters.

How do we refer to characters that are special themselves (like `.`, `*`, `-`, `[`, and `\`) when we mean them literally, not in their special usage? That is, if we are trying to match a period, or a star, or a bracket or paren? To get the literal meaning of a special character, we need to precede them with a backslash, (i.e., `r"\."`, `r"*"`, `r"\["`, and `r"\\"`).

| Regex | Match | First Patterns Matched |
|-----------------|-----------------|--|
| <code>*</code> | an asterisk “*” | “K_A*P*L*A*N” |
| <code>\.</code> | a period “.” | “Dr. Livingston, I presume” |
| <code>\?</code> | a question mark | “Why don’t they come and lend a hand?” |
| <code>\n</code> | a newline | |
| <code>\t</code> | a tab | |

Figure 2.14 Some characters that need to be escaped (via backslash).

2.6.7 Substitutions and Capture Groups

substitution

An important use of regular expressions is in **substitutions**, where we want to replace one string with another. Regular expression can help us specify the string to be replaced as well as the replacement. In Python we use the function `re.sub()` (similar functions exist in other languages and environments).

`re.sub(pattern, repl, string)` takes three arguments: a *pattern* to search for, a *replacement* to replace it with, and a *string* in which to do the search and replacing. We could for example change every instance of `cherry` to `apricot` in `string`:

```
re.sub(r"cherry", r"apricot", string)
```

Or we could convert to upper case all the instances of a particular name:

```
re.sub(r"janet", r"Janet", string)
```

More often, however, the substitution depends in a more complex way on the string that matched the *pattern*. For example, suppose we have a document in which all the dates are in US format (mm/dd/yyyy) and we want to change them into the format used in the EU and many other regions: (dd-mm-yyyy). The *pattern* `r"\d{2}/\d{2}/\d{4}"` will match a date. But how do we specify in the *replacement* that we want to swap the date and month values?

capture group

The tool in regular expression for this is the **capture group**. A capture group uses parentheses to capture (*store*) the values that we matched in the search, so we can reuse them in the replacement. We put a set of parentheses around the part of the *pattern* we want to capture, and it will get stored in a numbered group (groups are numbered from left to right). Then in the *repl*, we refer back to that group with a number command.

Consider the following expression:

```
re.sub(r"(\d{2})/(\d{2})/(\d{4})", r"\2-\1-\3", string)}
```

We've put parentheses (and) around the two month digits, the two day digits, and the four year digits, thus storing the first 2 digits in group 1, the second 2 digits in group 2, and the final digits in group 3. Then in the *repl* string, we use **number** operators \1, \2, and \3, to refer back to the first, second, and third registers. The result would take a string like

```
The date is 10/15/2011
```

and convert it to

```
The date is 15-10-2011
```

Capture groups can be useful even if we are not doing substitutions. For example we can use them to find repetitions, something we often need in text processing. For example, to find a repeated word in a string, we can use this pattern which searches for a word, captures it in a group, and then refers back to it after whitespace:

```
r"\b([A-Za-z]+)\s+\1\b"
```

Parentheses thus have a double function in regular expressions; they are used to group terms for specifying the order in which operators should apply, and they are used to capture the match. Occasionally we need parentheses for grouping, but don't want to capture the resulting pattern. In that case we use a **non-capturing group**, which is specified by putting the special commands ?: after the open parenthesis, in the form (?: pattern). Non-capture groups are usually used when we are trying to capture only part of a long or complex pattern. Perhaps we are matching a sequence of dates (\d\d/\d\d/\d\d\d\d) separated by spaces and we want to extract only the 15th one. We need to use parenthesis in order to use the counting operator on the first 14, but we don't want to store all the useless information. The following pattern only stores the 15th date in group 1:

```
r"(?:\d\d/\d\d/\d\d\d\d\s+){14}(\d\d/\d\d/\d\d\d\d)" (2.17)
```

Substitutions and capture groups are also useful for implementing historically important chatbots like ELIZA (Weizenbaum, 1966). Recall that ELIZA simulates a Rogerian psychologist by carrying on conversations like the following:

```
User2: They're always bugging us about something or other.
ELIZA2: CAN YOU THINK OF A SPECIFIC EXAMPLE
User3: Well, my boyfriend made me come here.
ELIZA3: YOUR BOYFRIEND MADE YOU COME HERE
User4: He says I'm depressed much of the time.
ELIZA4: I AM SORRY TO HEAR YOU ARE DEPRESSED
```

ELIZA works by having a series or cascade of regex substitutions each of which matches and changes some part of the input lines. After the input is uppercased, substitutions change all instances of MY to YOUR, and I'M to YOU ARE, and so on. That way when ELIZA repeats back part of the user utterance, it will seem to be referring correctly to the user. The next set of substitutions matches and replaces other patterns in the input, turning the input into a complete response. Here are some examples:

```
re.sub(r".* YOU ARE (DEPRESSED|SAD) .*",r"I AM SORRY TO HEAR YOU ARE \1",input)
re.sub(r".* YOU ARE (DEPRESSED|SAD) .*",r"WHY DO YOU THINK YOU ARE \1",input)
re.sub(r".* ALWAYS .*",r"CAN YOU THINK OF A SPECIFIC EXAMPLE",input)
```

CHAPTER

17

Sequence Labeling for Parts of Speech and Named Entities

To each word a warbling note
A Midsummer Night's Dream, VI

parts of speech

Dionysius Thrax of Alexandria (c. 100 B.C.), or perhaps someone else (it was a long time ago), wrote a grammatical sketch of Greek (a “*technē*”) that summarized the linguistic knowledge of his day. This work is the source of an astonishing proportion of modern linguistic vocabulary, including the words *syntax*, *diphthong*, *clitic*, and *analogy*. Also included are a description of eight **parts of speech**: noun, verb, pronoun, preposition, adverb, conjunction, participle, and article. Although earlier scholars (including Aristotle as well as the Stoics) had their own lists of parts of speech, it was Thrax’s set of eight that became the basis for descriptions of European languages for the next 2000 years. (All the way to the *Schoolhouse Rock* educational television shows of our childhood, which had songs about 8 parts of speech, like the late great Bob Dorough’s *Conjunction Junction*.) The durability of parts of speech through two millennia speaks to their centrality in models of human language.

named entity

Proper names are another important and anciently studied linguistic category. While parts of speech are generally assigned to individual words or morphemes, a proper name is often an entire multiword phrase, like the name “Marie Curie”, the location “New York City”, or the organization “Stanford University”. We’ll use the term **named entity** for, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization, although as we’ll see the term is commonly extended to include things that aren’t entities per se.

POS

Parts of speech (also known as **POS**) and named entities are useful clues to sentence structure and meaning. Knowing whether a word is a noun or a verb tells us about likely neighboring words (nouns in English are preceded by determiners and adjectives, verbs by nouns) and syntactic structure (verbs have dependency links to nouns), making part-of-speech tagging a key aspect of parsing. Knowing if a named entity like *Washington* is a name of a person, a place, or a university is important to many natural language processing tasks like question answering, stance detection, or information extraction.

In this chapter we’ll introduce the task of **part-of-speech tagging**, taking a sequence of words and assigning each word a part of speech like NOUN or VERB, and the task of **named entity recognition (NER)**, assigning words or phrases tags like PERSON, LOCATION, or ORGANIZATION.

sequence labeling

Such tasks in which we assign, to each word x_i in an input word sequence, a label y_i , so that the output sequence Y has the same length as the input sequence X are called **sequence labeling** tasks. We’ll introduce classic sequence labeling algorithms, one generative—the Hidden Markov Model (HMM)—and one discriminative—the Conditional Random Field (CRF). In following chapters we’ll introduce modern sequence labelers based on RNNs and Transformers.

17.1 (Mostly) English Word Classes

Until now we have been using part-of-speech terms like **noun** and **verb** rather freely. In this section we give more complete definitions. While word classes do have semantic tendencies—adjectives, for example, often describe *properties* and nouns *people*—parts of speech are defined instead based on their grammatical relationship with neighboring words or the morphological properties about their affixes.

| | Tag | Description | Example |
|--------------------|--|--|---------------------------------------|
| Open Class | ADJ | Adjective: noun modifiers describing properties | <i>red, young, awesome</i> |
| | ADV | Adverb: verb modifiers of time, place, manner | <i>very, slowly, home, yesterday</i> |
| | NOUN | words for persons, places, things, etc. | <i>algorithm, cat, mango, beauty</i> |
| | VERB | words for actions and processes | <i>draw, provide, go</i> |
| | PROPN | Proper noun: name of a person, organization, place, etc.. | <i>Regina, IBM, Colorado</i> |
| | INTJ | Interjection: exclamation, greeting, yes/no response, etc. | <i>oh, um, yes, hello</i> |
| Closed Class Words | ADP | Adposition (Preposition/Postposition): marks a noun's spacial, temporal, or other relation | <i>in, on, by, under</i> |
| | AUX | Auxiliary: helping verb marking tense, aspect, mood, etc., | <i>can, may, should, are</i> |
| | CCONJ | Coordinating Conjunction: joins two phrases/clauses | <i>and, or, but</i> |
| | DET | Determiner: marks noun phrase properties | <i>a, an, the, this</i> |
| | NUM | Numeral | <i>one, two, 2026, 11:00, hundred</i> |
| | PART | Particle: a function word that must be associated with another word | <i>'s, not, (infinitive) to</i> |
| | PRON | Pronoun: a shorthand for referring to an entity or event | <i>she, who, I, others</i> |
| SCONJ | Subordinating Conjunction: joins a main clause with a subordinate clause such as a sentential complement | <i>whether, because</i> | |
| Other | PUNCT | Punctuation | <i>; , ()</i> |
| | SYM | Symbols like \$ or emoji | <i>\$, %</i> |
| | X | Other | <i>asdf, qwfg</i> |

Figure 17.1 The 17 parts of speech in the Universal Dependencies tagset (de Marneffe et al., 2021). Features can be added to make finer-grained distinctions (with properties like number, case, definiteness, and so on).

closed class Parts of speech fall into two broad categories: **closed class** and **open class**.

open class Closed classes are those with relatively fixed membership, such as prepositions—new prepositions are rarely coined. By contrast, nouns and verbs are open classes—new nouns and verbs like *iPhone* or *to fax* are continually being created or borrowed.

function word Closed class words are generally **function words** like *of, it, and, or you*, which tend to be very short, occur frequently, and often have structuring uses in grammar.

Four major open classes occur in the languages of the world: **nouns** (including proper nouns), **verbs**, **adjectives**, and **adverbs**, as well as the smaller open class of **interjections**. English has all five, although not every language does.

noun **Nouns** are words for people, places, or things, but include others as well. **Common noun** **Common nouns** include concrete terms like *cat* and *mango*, abstractions like *algorithm* and *beauty*, and verb-like terms like *pacing* as in *His pacing to and fro became quite annoying*. Nouns in English can occur with determiners (*a goat, this bandwidth*) take possessives (*IBM's annual revenue*), and may occur in the plural (*goats, abaci*).

count noun Many languages, including English, divide common nouns into **count nouns** and **mass nouns**. Count nouns can occur in the singular and plural (*goat/goats, relationship/relationships*) and can be counted (*one goat, two goats*). Mass nouns are used when something is conceptualized as a homogeneous group. So *snow, salt, and communism* are not counted (i.e., **two snows* or **two communisms*).

proper noun **Proper nouns**, like *Regina, Colorado, and IBM*, are names of specific persons or entities.

| | |
|------------------------|--|
| verb | <p>Verbs refer to actions and processes, including main verbs like <i>draw</i>, <i>provide</i>, and <i>go</i>. English verbs have inflections (non-third-person-singular (<i>eat</i>), third-person-singular (<i>eats</i>), progressive (<i>eating</i>), past participle (<i>eaten</i>)). While many scholars believe that all human languages have the categories of noun and verb, others have argued that some languages, such as Riau Indonesian and Tongan, don't even make this distinction (Broschart 1997; Evans 2000; Gil 2000).</p> |
| adjective | <p>Adjectives often describe properties or qualities of nouns, like color (<i>white</i>, <i>black</i>), age (<i>old</i>, <i>young</i>), and value (<i>good</i>, <i>bad</i>), but there are languages without adjectives. In Korean, for example, the words corresponding to English adjectives act as a subclass of verbs, so what is in English an adjective “beautiful” acts in Korean like a verb meaning “to be beautiful”.</p> |
| adverb | <p>Adverbs are a hodge-podge. All the italicized words in this example are adverbs:</p> <p style="text-align: center;"><i>Actually</i>, I ran <i>home</i> <i>extremely</i> <i>quickly</i> <i>yesterday</i></p> <p>Adverbs generally modify something (often verbs, hence the name “adverb”, but also other adverbs and entire verb phrases). Directional adverbs or locative adverbs (<i>home</i>, <i>here</i>, <i>downhill</i>) specify the direction or location of some action; degree adverbs (<i>extremely</i>, <i>very</i>, <i>somewhat</i>) specify the extent of some action, process, or property; manner adverbs (<i>slowly</i>, <i>slinkily</i>, <i>delicately</i>) describe the manner of some action or process; and temporal adverbs describe the time that some action or event took place (<i>yesterday</i>, <i>Monday</i>).</p> |
| locative degree | |
| manner temporal | |
| interjection | <p>Interjections (<i>oh</i>, <i>hey</i>, <i>alas</i>, <i>uh</i>, <i>um</i>) are a smaller open class that also includes greetings (<i>hello</i>, <i>goodbye</i>) and question responses (<i>yes</i>, <i>no</i>, <i>uh-huh</i>).</p> |
| preposition | <p>English adpositions occur before nouns, hence are called prepositions. They can indicate spatial or temporal relations, whether literal (<i>on it</i>, <i>before then</i>, <i>by the house</i>) or metaphorical (<i>on time</i>, <i>with gusto</i>, <i>beside herself</i>), and relations like marking the agent in <i>Hamlet was written by Shakespeare</i>.</p> |
| particle | <p>A particle resembles a preposition or an adverb and is used in combination with a verb. Particles often have extended meanings that aren't quite the same as the prepositions they resemble, as in the particle <i>over</i> in <i>she turned the paper over</i>. A verb and a particle acting as a single unit is called a phrasal verb. The meaning of phrasal verbs is often non-compositional—not predictable from the individual meanings of the verb and the particle. Thus, <i>turn down</i> means ‘reject’, <i>rule out</i> ‘eliminate’, and <i>go on</i> ‘continue’.</p> |
| phrasal verb | |
| determiner article | <p>Determiners like <i>this</i> and <i>that</i> (<i>this chapter</i>, <i>that page</i>) can mark the start of an English noun phrase. Articles like <i>a</i>, <i>an</i>, and <i>the</i>, are a type of determiner that mark discourse properties of the noun and are quite frequent; <i>the</i> is the most common word in written English, with <i>a</i> and <i>an</i> right behind.</p> |
| conjunction | <p>Conjunctions join two phrases, clauses, or sentences. Coordinating conjunctions like <i>and</i>, <i>or</i>, and <i>but</i> join two elements of equal status. Subordinating conjunctions are used when one of the elements has some embedded status. For example, the subordinating conjunction <i>that</i> in “<i>I thought that you might like some milk</i>” links the main clause <i>I thought</i> with the subordinate clause <i>you might like some milk</i>. This clause is called subordinate because this entire clause is the “content” of the main verb <i>thought</i>. Subordinating conjunctions like <i>that</i> which link a verb to its argument in this way are also called complementizers.</p> |
| complementizer pronoun | |
| wh | <p>Pronouns act as a shorthand for referring to an entity or event. Personal pronouns refer to persons or entities (<i>you</i>, <i>she</i>, <i>I</i>, <i>it</i>, <i>me</i>, etc.). Possessive pronouns are forms of personal pronouns that indicate either actual possession or more often just an abstract relation between the person and some object (<i>my</i>, <i>your</i>, <i>his</i>, <i>her</i>, <i>its</i>, <i>one's</i>, <i>our</i>, <i>their</i>). Wh-pronouns (<i>what</i>, <i>who</i>, <i>whom</i>, <i>whoever</i>) are used in certain question</p> |

forms, or act as complementizers (*Frida, who married Diego...*).

auxiliary **Auxiliary** verbs mark semantic features of a main verb such as its tense, whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, or desired (mood). English auxiliaries include the

copula **copula** verb *be*, the two verbs *do* and *have*, forms, as well as **modal verbs** used to

modal mark the mood associated with the event depicted by the main verb: *can* indicates ability or possibility, *may* permission or possibility, *must* necessity.

An English-specific tagset, the Penn Treebank tagset (Marcus et al., 1993), shown in Fig. 17.2, has been used to label many syntactically annotated corpora like the Penn Treebank corpora, so it is worth knowing about.

| Tag | Description | Example | Tag | Description | Example | Tag | Description | Example |
|-----|-------------------------------|---------------------|-------|--------------------|--------------------|------|----------------------|--------------------|
| CC | coord. conj. | <i>and, but, or</i> | NNP | proper noun, sing. | <i>IBM</i> | TO | infinitive to | <i>to</i> |
| CD | cardinal number | <i>one, two</i> | NNPS | proper noun, plu. | <i>Carolinas</i> | UH | interjection | <i>ah, oops</i> |
| DT | determiner | <i>a, the</i> | NNS | noun, plural | <i>llamas</i> | VB | verb base | <i>eat</i> |
| EX | existential 'there' | <i>there</i> | PDT | predeterminer | <i>all, both</i> | VBD | verb past tense | <i>ate</i> |
| FW | foreign word | <i>mea culpa</i> | POS | possessive ending | <i>'s</i> | VBG | verb gerund | <i>eating</i> |
| IN | preposition/ subordin-conj | <i>of, in, by</i> | PRP | personal pronoun | <i>I, you, he</i> | VBN | verb past participle | <i>eaten</i> |
| JJ | adjective | <i>yellow</i> | PRP\$ | possess. pronoun | <i>your</i> | VBP | verb non-3sg-pr | <i>eat</i> |
| JJR | comparative adj | <i>bigger</i> | RB | adverb | <i>quickly</i> | VBZ | verb 3sg pres | <i>eats</i> |
| JJS | superlative adj | <i>wildest</i> | RBR | comparative adv | <i>faster</i> | WDT | wh-determ. | <i>which, that</i> |
| LS | list item marker | <i>1, 2, One</i> | RBS | superlatv. adv | <i>fastest</i> | WP | wh-pronoun | <i>what, who</i> |
| MD | modal | <i>can, should</i> | RP | particle | <i>up, off</i> | WP\$ | wh-possess. | <i>whose</i> |
| NN | sing or mass noun | <i>llama</i> | SYM | symbol | <i>+, %, &</i> | WRB | wh-adverb | <i>how, where</i> |

Figure 17.2 Penn Treebank core 36 part-of-speech tags.

Below we show some examples with each word tagged according to both the UD (in blue) and Penn (in red) tagsets. Notice that the Penn tagset distinguishes tense and participles on verbs, and has a special tag for the existential *there* construction in English. Note that since *London Journal of Medicine* is a proper noun, both tagsets mark its component nouns as PROPN/NNP, including *journal* and *medicine*, which might otherwise be labeled as common nouns (NOUN/NN).

(17.1) There/PRON/EX are/VERB/VBP 70/NUM/CD children/NOUN/NNS
there/ADV/RB ./PUNC/.

(17.2) Preliminary/ADJ/JJ findings/NOUN/NNS were/AUX/VBD
reported/VERB/VBN in/ADP/IN today/NOUN/NN 's/PART/POS
London/PROPN/NNP Journal/PROPN/NNP of/ADP/IN Medicine/PROPN/NNP

17.2 Part-of-Speech Tagging

part-of-speech tagging

Part-of-speech tagging is the process of assigning a part-of-speech to each word in a text. The input is a sequence x_1, x_2, \dots, x_n of (tokenized) words and a tagset, and the output is a sequence y_1, y_2, \dots, y_n of tags, each output y_i corresponding exactly to one input x_i , as shown in the intuition in Fig. 17.3.

ambiguous

Tagging is a **disambiguation** task; words are **ambiguous**—have more than one possible part-of-speech—and the goal is to find the correct tag for the situation. For example, *book* can be a verb (*book that flight*) or a noun (*hand me that book*). *That* can be a determiner (*Does that flight serve dinner*) or a complementizer (*I*

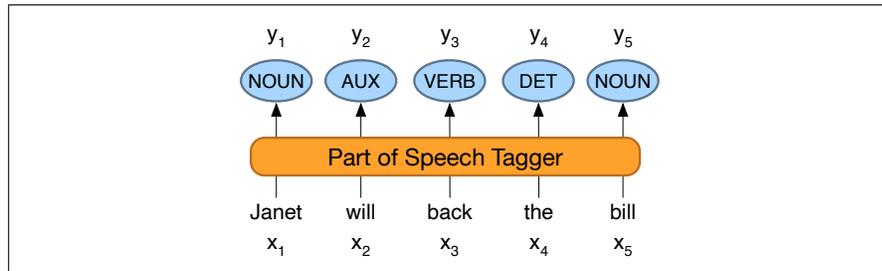


Figure 17.3 The task of part-of-speech tagging: mapping from input words x_1, x_2, \dots, x_n to output POS tags y_1, y_2, \dots, y_n .

ambiguity
resolution

thought that your flight was earlier). The goal of POS-tagging is to **resolve** these ambiguities, choosing the proper tag for the context.

accuracy

The **accuracy** of part-of-speech tagging algorithms (the percentage of test set tags that match human gold labels) is extremely high. One study found accuracies over 97% across 15 languages from the Universal Dependency (UD) treebank (Wu and Dredze, 2019). Accuracies on various English treebanks are also 97% (no matter the algorithm; HMMs, CRFs, BERT perform similarly). This 97% number is also about the human performance on this task, at least for English (Manning, 2011).

| Types: | WSJ | Brown |
|----------------------------|---------------|---------------|
| Unambiguous (1 tag) | 44,432 (86%) | 45,799 (85%) |
| Ambiguous (2+ tags) | 7,025 (14%) | 8,050 (15%) |
| Tokens: | | |
| Unambiguous (1 tag) | 577,421 (45%) | 384,349 (33%) |
| Ambiguous (2+ tags) | 711,780 (55%) | 786,646 (67%) |

Figure 17.4 Tag ambiguity in the Brown and WSJ corpora (Treebank-3 45-tag tagset).

We'll introduce algorithms for the task in the next few sections, but first let's explore the task. Exactly how hard is it? Fig. 17.4 shows that most word types (85-86%) are unambiguous (*Janet* is always NNP, *hesitantly* is always RB). But the ambiguous words, though accounting for only 14-15% of the vocabulary, are very common, and 55-67% of word tokens in running text are ambiguous. Particularly ambiguous common words include *that*, *back*, *down*, *put* and *set*; here are some examples of the 6 different parts of speech for the word *back*:

earnings growth took a **back/JJ** seat
 a small building in the **back/NN**
 a clear majority of senators **back/VBP** the bill
 Dave began to **back/VB** toward the door
 enable the country to buy **back/RP** debt
 I was twenty-one **back/RB** then

Nonetheless, many words are easy to disambiguate, because their different tags aren't equally likely. For example, *a* can be a determiner or the letter *a*, but the determiner sense is much more likely.

This idea suggests a useful **baseline**: given an ambiguous word, choose the tag which is **most frequent** in the training corpus. This is a key concept:

Most Frequent Class Baseline: Always compare a classifier against a baseline at least as good as the most frequent class baseline (assigning each token to the class it occurred in most often in the training set).

The most-frequent-tag baseline has an accuracy of about 92%¹. The baseline thus differs from the state-of-the-art and human ceiling (97%) by only 5%.

17.3 Named Entities and Named Entity Tagging

Part of speech tagging can tell us that words like *Janet*, *Stanford University*, and *Colorado* are all proper nouns; being a proper noun is a grammatical property of these words. But viewed from a semantic perspective, these proper nouns refer to different kinds of entities: Janet is a person, Stanford University is an organization, and Colorado is a location.

named entity

Here we re-introduce the concept of a **named entity**, which was also introduced in Section ?? for readers who haven't yet read Chapter 9.

named entity

A **named entity** is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. The task of **named entity recognition (NER)** is to find spans of text that constitute proper names and tag the type of the entity. Four entity tags are most common: **PER** (person), **LOC** (location), **ORG** (organization), or **GPE** (geo-political entity). However, the term **named entity** is commonly extended to include things that aren't entities per se, including dates, times, and other kinds of temporal expressions, and even numerical expressions like prices. Here's an example of the output of an NER tagger:

named entity
recognition
NER

Citing high fuel prices, [ORG United Airlines] said [TIME Friday] it has increased fares by [MONEY \$6] per round trip on flights to some cities also served by lower-cost carriers. [ORG American Airlines], a unit of [ORG AMR Corp.], immediately matched the move, spokesman [PER Tim Wagner] said. [ORG United], a unit of [ORG UAL Corp.], said the increase took effect [TIME Thursday] and applies to most routes where it competes against discount carriers, such as [LOC Chicago] to [LOC Dallas] and [LOC Denver] to [LOC San Francisco].

The text contains 13 mentions of named entities including 5 organizations, 4 locations, 2 times, 1 person, and 1 mention of money. Figure 17.5 shows typical generic named entity types. Many applications will also need to use specific entity types like proteins, genes, commercial products, or works of art.

| Type | Tag | Sample Categories | Example sentences |
|----------------------|-----|--------------------------|---|
| People | PER | people, characters | Turing is a giant of computer science. |
| Organization | ORG | companies, sports teams | The IPCC warned about the cyclone. |
| Location | LOC | regions, mountains, seas | Mt. Sanitas is in Sunshine Canyon . |
| Geo-Political Entity | GPE | countries, states | Palo Alto is raising the fees for parking. |

Figure 17.5 A list of generic named entity types with the kinds of entities they refer to.

Named entity tagging is a useful first step in lots of natural language processing tasks. In sentiment analysis we might want to know a consumer's sentiment toward a particular entity. Entities are a useful first stage in question answering, or for linking text to information in structured knowledge sources like Wikipedia. And named entity tagging is also central to tasks involving building semantic representations, like extracting events and the relationship between participants.

¹ In English, on the WSJ corpus, tested on sections 22-24.

Unlike part-of-speech tagging, where there is no segmentation problem since each word gets one tag, the task of named entity recognition is to find and label *spans* of text, and is difficult partly because of the ambiguity of segmentation; we need to decide what’s an entity and what isn’t, and where the boundaries are. Indeed, most words in a text will not be named entities. Another difficulty is caused by type ambiguity. The mention JFK can refer to a person, the airport in New York, or any number of schools, bridges, and streets around the United States. Some examples of this kind of cross-type confusion are given in Figure 17.6.

[PER Washington] was born into slavery on the farm of James Burroughs.
 [ORG Washington] went up 2 games to 1 in the four-game series.
 Blair arrived in [LOC Washington] for what may well be his last state visit.
 In June, [GPE Washington] passed a primary seatbelt law.

Figure 17.6 Examples of type ambiguities in the use of the name *Washington*.

The standard approach to sequence labeling for a span-recognition problem like NER is **BIO** tagging (Ramshaw and Marcus, 1995). This is a method that allows us to treat NER like a word-by-word sequence labeling task, via tags that capture both the boundary and the named entity type. Consider the following sentence:

[PER Jane Villanueva] of [ORG United], a unit of [ORG United Airlines Holding], said the fare applies to the [LOC Chicago] route.

BIO Figure 17.7 shows the same excerpt represented with **BIO** tagging, as well as variants called **IO** tagging and **BIOES** tagging. In **BIO** tagging we label any token that *begins* a span of interest with the label **B**, tokens that occur *inside* a span are tagged with an **I**, and any tokens *outside* of any span of interest are labeled **O**. While there is only one **O** tag, we’ll have distinct **B** and **I** tags for each named entity class. The number of tags is thus $2n + 1$ tags, where n is the number of entity types. **BIO** tagging can represent exactly the same information as the bracketed notation, but has the advantage that we can represent the task in the same simple sequence modeling way as part-of-speech tagging: assigning a single label y_i to each input word x_i :

| Words | IO Label | BIO Label | BIOES Label |
|------------|----------|-----------|-------------|
| Jane | I-PER | B-PER | B-PER |
| Villanueva | I-PER | I-PER | E-PER |
| of | O | O | O |
| United | I-ORG | B-ORG | B-ORG |
| Airlines | I-ORG | I-ORG | I-ORG |
| Holding | I-ORG | I-ORG | E-ORG |
| discussed | O | O | O |
| the | O | O | O |
| Chicago | I-LOC | B-LOC | S-LOC |
| route | O | O | O |
| . | O | O | O |

Figure 17.7 NER as a sequence model, showing IO, BIO, and BIOES taggings.

We’ve also shown two variant tagging schemes: **IO** tagging, which loses some information by eliminating the **B** tag, and **BIOES** tagging, which adds an end tag **E** for the end of a span, and a span tag **S** for a span consisting of only one word. A sequence labeler (HMM, CRF, RNN, Transformer, etc.) is trained to label each token in a text with tags that indicate the presence (or absence) of particular kinds of named entities.