



Declarative Programming

Types of Programming

Imperative programming: Describe what you want a computer to do

Often involves mutation for the purpose of computing a result.

Computational efficiency is often determined by the details of the program.

E.g., object-oriented programming is a useful way of organizing imperative programs.

Declarative programming: Describe the result you want a computer to produce

Often abstracts away the details of how memory is changing during computation.

Computational efficiency is often determined by the interpreter or language.

E.g., functional programming describes a result using function composition.

_

Types of Programming Languages

General-purpose languages: Designed to describe any computation

Python, Scheme, Javascript, Java, C, C++, etc.

Languages differ in the programming styles that they promote.

Language features make some languages more suitable to certain applications.

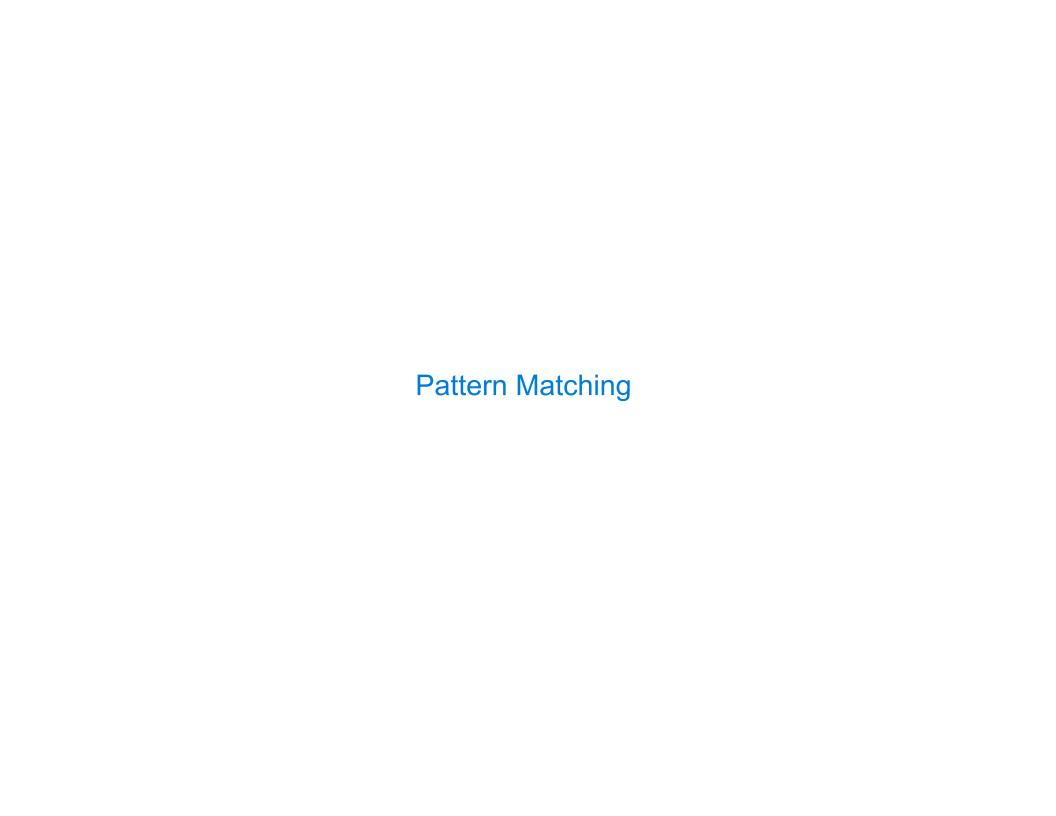
Domain-specific languages: Designed to solve particular classes of problems

SQL, HTML, CSS, regular expressions, etc.

Often declarative in character: the language describes what to compute/create, not how.

Often embedded into general-purpose languages.

J



Pattern Matching in Strings

```
Let's pretend that an email address is any string of the form <name>@<domain> where:
• All the characters are letters, numbers, '@', '.', or '_'.
• There is exactly one @.

    <domain> has no .. and ends in .<tld>, where <tld> is exactly three letters.

E.g., 'oski@berkeley.edu' and 'oski 4ever@cs.berkeley.edu' are allowed, but not:
'oski@berkeley'
'oski@berkeley.info'
'oski@berkeley.3du'
'oski!@berkeley.edu'
'oski @berkeley.edu'
'oski@berkeley..edu'
                                             (Demo)
```

The rules for actual email addresses are more complicated: https://datatracker.ietf.org/doc/html/rfc822

Pattern Matching Using Regular Expressions

(Demo)

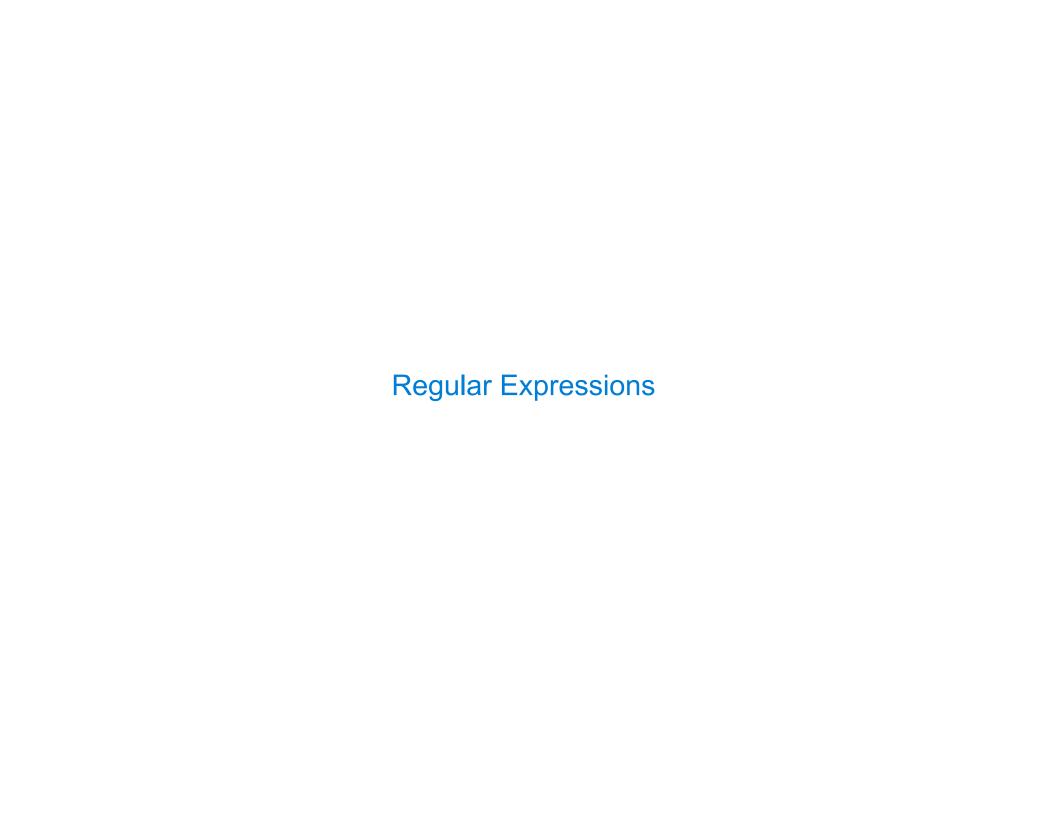
a word followed by . (e.g., berkeley.)

a letter or number (or _) $\frac{}{W+Q}(\overline{)W+})+\overline{[A-Za-z]}\{3\}$ one or more letters/numbers

one or more parts of a domain name ending in .

A regular expression describes a string pattern from left to right:

- A character class such as \w, @, or [A-Za-z] describes which individual characters match
- A quantifier such as + or {3} describes repetition
- Parentheses describe groups, which correspond to substrings.



Matching Individual Characters

Except for special characters, a single character in a regex matches itself in a string.

B matches B

A sequence of characters in a regex matches that same sequence in a string.

Berkeley matches Berkeley

Special characters are: $\ \ (\)\ [\]\ \{\ \}\ +\ *\ ?\ |\ $.$

To match a special character, it must be escaped in the regex by placing a \ before it.

\(\\\._\./\) matches (\._./)

Character Categories

1	Matches any character	.a.	cal, ha!, (a)
\w	Matches letters, numbers or _	\wa\w	cal, dad, 3am
/d	Matches a digit	\d\d	61, 00
\s	Matches a whitespace character (space, tab, newline)	\d\s\d	1 2
[]	Encloses a list of options or ranges	b[aeiou]d	bad, bed, bid, bod, bud

A character class expression [...] can contain \d and \w and ranges such as 0-5.

[a-s\d]+ matches cs61a

11

Groups

Groups, which are surrounded by parentheses, have several purposes.

They correspond to substrings, and matching the whole pattern also matches each group

Fall 20(\d\d) matches Fall 2021 and the group matches 21

The | character matches either of two sequences

(Fall|Spring) 20(\d\d) matches either Fall 2021 or Spring 2021

A whole group can be repeated multiple times

l(ol)+ matches lol and lolol and lololol but not lolo

Quantifiers

A quantifier expression $(*, +, ?, \{...\})$ applies to the previous group or the previous character if there is no group

lo[ol]+ matches lol or lolol or looool or lolllll

+	One or more copies	aw+	awwww
*	Zero or more copies	b[a-z]*y	by, buy, buoy, berkeley
?	Zero or one copy	:[-o]?\)	:) :0) :-)
{min, max}	A particular number of copies or a range	ya{2,4}y	yaay, yaaay, yaaaay

If a range has only one number, then it is both the min and max.

B(e..){2}y matches Berkeley

Anchors

A common application of regular expressions is to search for a pattern within a string.

Anchors describe the context within a longer string that a pattern can appear.

For example, consider the following string in which we will search:

Tell Oski that he lost his hat

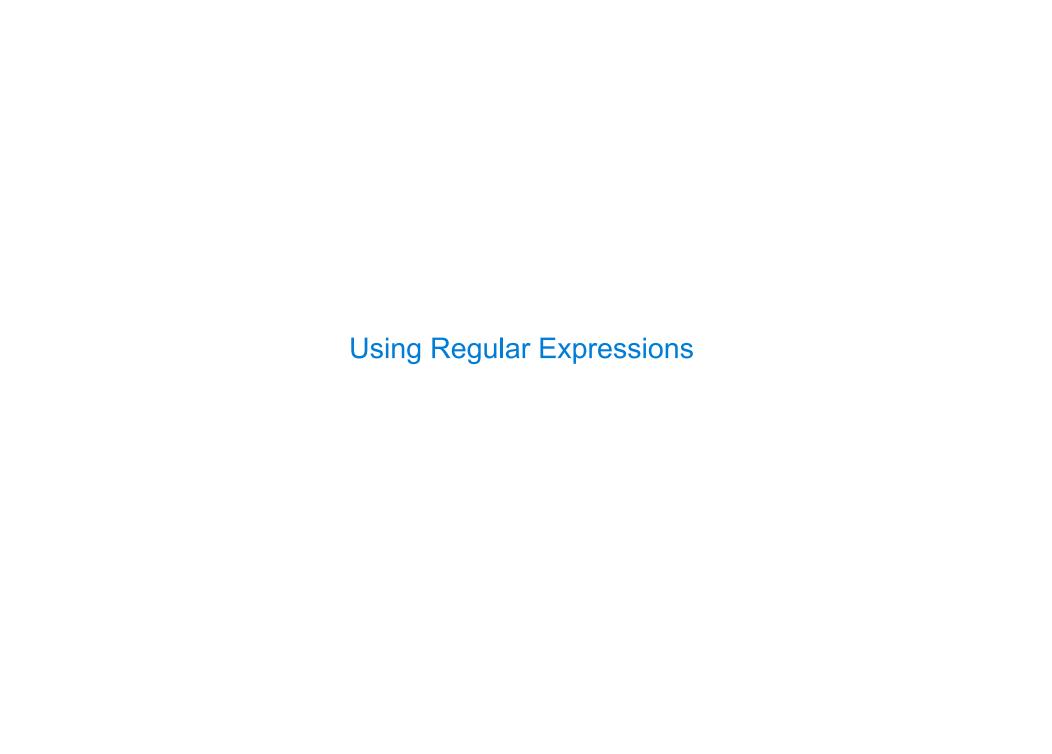
The ^ and \$ anchors correspond to the start and end of the full string

^\w+ matches **Tell** (but not Oski)

.t\$ matches at (but not st)

The \b anchor corresponds to the beginning or end of a word

.s\b matches is (bot not 0s)



Regular Expressions in Python Programs

The re module has search, fullmatch, match, findall, sub, and more.

```
def email(s):
    return bool(re.fullmatch(r'\w+@(\w+\.)+[A-Za-z]{3}', s))
```

When writing a regular expression in Python, use a raw string preceded by r to stop Python from treating \ as an escape character.

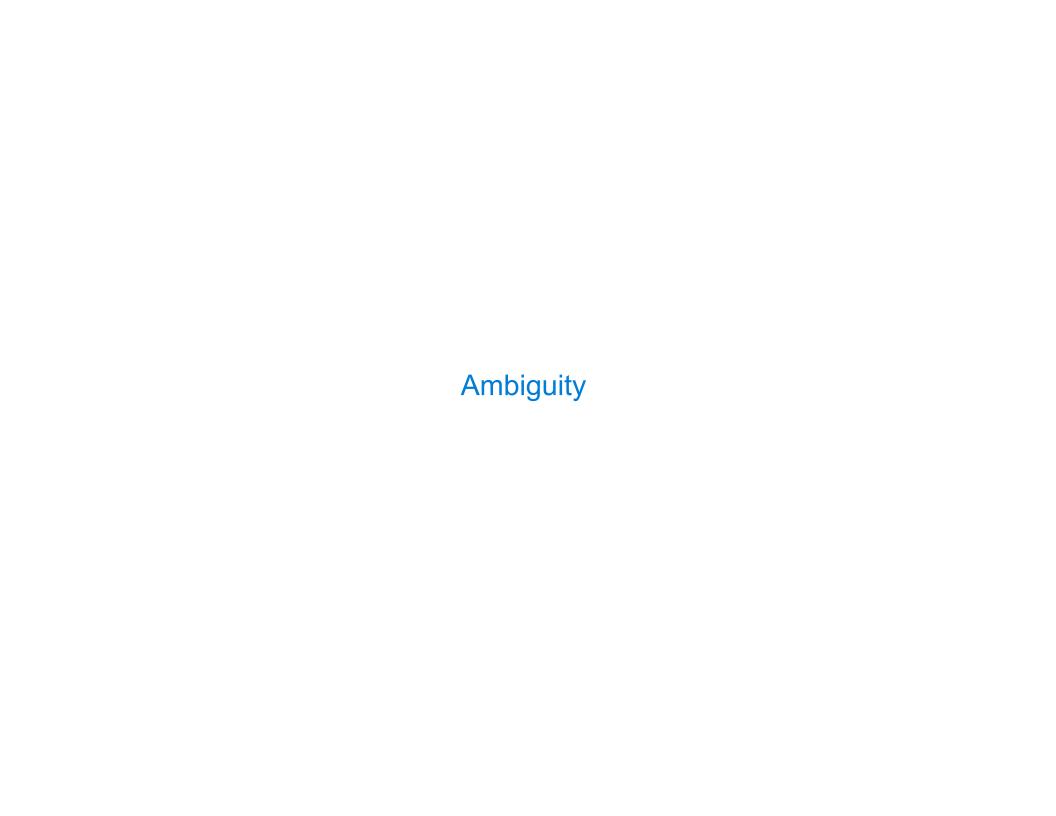
flags allow you to control, for example, whether matching can include multiple lines.

fullmatch(pattern, string, flags=0)

Try to apply the pattern to all of the string, returning a Match object, or None if no match was found.

A Match object gives access to the substrings that match groups within the regex.

(Demo)



Rules for Ambiguous Matches

```
re.search returns the first match within a string.
```

Quantifiers are matched greedily, meaning that a longer variant will be preferred.

```
>>> re.search(r'Cal(ifornia)?', 'Is California known as Cal?')
<re.Match object; span=(3, 13), match='California'>
>>> re.search(r'Cal.*a', 'Is California known as Cal?')
<re.Match object; span=(3, 25), match='California known as Ca'>
```

Each quantified expression is matched to the longest possible substring from left to right.

```
>>> re.search(r'Cal(\w*i)\w*', 'Is California known as Cal?').group(1)
'iforni'
```

The choice to the left of | is preferred.

```
>>> re.search(r'Cal|California', 'Is California known as Cal?')
<re.Match object; span=(3, 6), match='Cal'>
```

Lazy Quantification

Lazy operators *?, +?, and ?? correspond to *, +, and ?, but match as little as possible.

Changing greedy operators (*, +, ?) to lazy operators cannot change whether a regular expression matches a string, but it can affect the substrings that are matched by groups or the whole expression.

```
>>> re.search(r'Cal(ifornia)?', 'Is California known as Cal?')
<re.Match object; span=(3, 13), match='California'>
>>> re.search(r'Cal.*a', 'Is California known as Cal?')
<re.Match object; span=(3, 25), match='California known as Cal?')
<re.Match object; span=(3, 6), match='California known as Cal?')
<re.Match object; span=(3, 6), match='California known as Cal?')
>>> re.search(r'Cal.*?a', 'Is California known as Cal?')
<re.Match object; span=(3, 13), match='California'>
```