Regular Expressions
Announcements
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.
Pattern Matching
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.)

\[ \text{\w+@}(\text{\w+\.})+\text{[A-Za-z]}\{3\} \]

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.
Regular Expressions
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

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

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

[a-s\d]+ matches cs61a
Groups

Groups, which are surrounded by parentheses, have several purposes. They correspond to substrings, and matching the whole pattern also matches each group.

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

The | character matches either of two sequences

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

A whole group can be repeated multiple times

\textbf{l(ol)+} matches \textbf{lol} and \textbf{lolol} and \textbf{lololol} but not \textbf{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 **loooool** or **lolllll**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>One or more copies</td>
<td>aw+</td>
<td>awwwwww</td>
</tr>
<tr>
<td>*</td>
<td>Zero or more copies</td>
<td>b[a-z]*y</td>
<td>by, buy, buoy, berkeley</td>
</tr>
<tr>
<td>?</td>
<td>Zero or one copy</td>
<td>[:o]?)</td>
<td>:) :o) :-)</td>
</tr>
<tr>
<td>{min, max}</td>
<td>A particular number of copies or a range</td>
<td>ya{2,4}y</td>
<td>yaay, yaaay, yaaaay</td>
</tr>
</tbody>
</table>

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 Os)
Using Regular Expressions
Regular Expressions in Python Programs

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

```python
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.

`fullmatch` is used 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)
Ambiguity
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.

```python
>>> re.search(r'Cal(iforni)?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.

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

The choice to the left of | is preferred.

```python
>>> 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.

```python
>>> 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'>

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