Lecture 31: Regular Expressions
Pattern Matching

- Programs that manipulate text often have a need to search a string for things other than simple substrings.

- For example: “Find all integer numerals in this string” or “Find all Scheme tokens in this program text.”

- Another application might be to check input: “Does this user’s response have the proper form?”

- Numerous programming languages provide some kind of pattern-matching facility to do this sort of thing.

- We can think of this as a kind of declarative programming, because the programmer is saying, e.g., “find somethin that looks like this” rather than “search for the substring ‘(’, then look for a ‘)’ after that” to check for a parenthesized expression.

- It’s up to library code to figure out how to find convert “looks like” into actual steps to search for that condition.
Regular Expressions

• One of the most widely available and useful mechanisms is the regular expression.

• Formally, regular expressions denote sets of strings that are called regular languages.

• But normally, we think of them as patterns that match certain strings.

• In Python, we denote them with strings and use them as patterns by means of functions and classes in the module re.

• Examples:

```python
import re

re.search('aardvark', S)  # Does S contain the substring "aardvark"?
re.fullmatch('[-+]?[0-7]+', S)  # Is all of S a properly formed signed octal number?
re.match(r'\s*[-+]?\d+', S)  # Does S start with a decimal number, possibly preceded by whitespace?
```
Small Preliminary: Raw Strings

• Traditionally, the backslash character (\) is often used in patterns.

• This can conflict with the usual Python string escape sequences (which begin with backslashes).

• For example, the two-character sequence \b matches the beginning or end of a word as a pattern, but in a string literal, it means an ASCII BEL, a single character that is supposed to be rendered as a noise.

• Furthermore, the string literal "\s" is supposed to match whitespace in a pattern, but various versions of Python treat it in inconsistent ways (it’s supposed to be an error in Python 3.9, where it should be written "\\s", as in Java.)

• So early on, Python introduced raw strings, which have an ‘r’ in front of the quotes, as in r"\s".

• In these strings, backslashes are just backslashes (except, annoyingly, that they cannot appear alone at the end of a string.)

• So generally, we use raw strings to denote patterns in Python.
Raw String Examples

```python
>>> "\n"
'\n'
>>> r"\n"
'\\n'
>>> print("I have\na newline in me."")
I have
a newline in me
>>> print(r"I have\na newline in me.")
I have\na newline in me.
```
Literal Characters

- (Sub)patterns that *don’t* contain any of the special characters
  \ ( ) [ ] { } + * ? | $ ^ .
  simply match themselves.

- Example: `r"Berkeley, CA 94720"` matches exactly the string or
  substring "Berkeley, CA 94720".

- To match one of the special characters above, precede with a backslash.
  Example: `r"\(1\+3\)"` matches exactly "(1+3)".
**Character Classes**

- A pattern of the form \([c_1c_2c_3\ldots]\), where each \(c_i\) is a character, is called a **character class** and matches any one of the characters \(c_i\).

- The special characters from before, other than backslash, carat, and \('\)', lose their special meanings.

- Inside a character class, \(c_1-c_2\) is short for all the characters between \(c_1\) and \(c_2\), inclusive. To include ‘-‘, put it first.

- **Examples:**
  
  \[
  [a\text{b},()]
  \]  
  matches any of 'a', 'b', '(', and ')' parentheses.

  \[
  [a-zA-Z0-9]
  \]  
  matches any (ASCII) letter or digit

  \[
  [-+0-9]
  \]  
  matches +, -, or any digit

- A character class of the form \([\sim c_1c_2c_3\ldots]\) (with a carat at the beginning) matches any one character that **isn’t** one of the characters \(c_i\). To include a carat in a character class, don’t put it first.

- **Example:**

  \[
  [\sim a-z]
  \]  
  matches any character except a lower-case letter
A Few Other Basic Patterns

These are not complete descriptions. They assume ASCII strings.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>. (dot)</td>
<td>Any single character, except newline or carriage return.</td>
</tr>
<tr>
<td>\d</td>
<td>Any single digit (same as [0-9].)</td>
</tr>
<tr>
<td>\s</td>
<td>Any single whitespace character: space, tab, newline, carriage return, &quot;\f&quot;, or &quot;\v&quot;</td>
</tr>
<tr>
<td>\S</td>
<td>Any single character that is not whitespace.</td>
</tr>
<tr>
<td>\w</td>
<td>Any single letter, digit, or underscore.</td>
</tr>
<tr>
<td>\W</td>
<td>Anything \w does not match.</td>
</tr>
</tbody>
</table>
Combining Patterns

• Just as arithmetic expressions have arithmetic operators, regular expression patterns also have a few operators.

• Some useful ones, in order of decreasing precedence. Here, $P$, $P_1$ and $P_2$ are patterns to be operated upon.
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
</table>
| $P_1P_2$     | A match for $P_1$ followed immediately by one for $P_2$. E.g., `r"ab[.,]"` matches "ab." or "ab,"
| $P*$         | 0 or more occurrences of $P$. E.g, `r"[a-z]*"` matches any sequence of lower-case letters or the empty string. |
| $P+$         | 1 or more occurrences of $P$. E.g, `r"\d+"` matches any non-empty sequence of digits.            |
| $P?$         | Matches either what $P$ does or the empty string. E.g., `r"[-+]?"` matches an optional sign.      |
| $P_1|P_2$       | Matches anything that either $P_1$ or $P_2$ does. E.g., `r"\d+|Inf"` matches either a decimal numeral or "Inf"
| $(P)$        | Matches whatever $P$ does. Parentheses group, just as in arithmetic expressions.                   |
## Anchors

A few patterns match the empty string, but only at certain places.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Normally matches the empty string at the beginning of a string.</td>
</tr>
<tr>
<td>$</td>
<td>Normally matches the empty string at the end of a string or just before a newline at the end of a string.</td>
</tr>
<tr>
<td>\b</td>
<td>Matches the empty string at the beginning or end of a word (composed of matches to \w).</td>
</tr>
<tr>
<td>\B</td>
<td>Matches the empty string where \b does not match.</td>
</tr>
</tbody>
</table>
Using Patterns in Python

• The methods `re.match`, `re.search`, and `re.fullmatch` all take a string containing a regular expression and a string of text. They return either a `match object` or, if there is no match, `None`.

• Match objects are 'true' values as far as Python is concerned, so one can use the results of these functions as True/False values:

```python
>>> for x in ("jack", "25", "-5", "aardvark"):
...     if re.fullmatch(r'-?\d+', x): print(f"{x} is a number")
25 is a number
-5 is a number
>>> bool(re.fullmatch(r'-?\d+', '123'))
True
>>> bool(re.fullmatch(r'-?\d+', '123 people'))
False
```
The Matching Methods

- **re.fullmatch** requires that the pattern match the entire searched string.

- **re.match** does not require that the whole string be matched, but does require that the matching string occur at the beginning of the string.

- **re.search** finds the first occurrence of the pattern anywhere in the string.

```python
>>> x = 'The Mill on the Floss.'
>>> bool(re.match(r'The', x))
True
>>> bool(re.fullmatch(r'The', x))
False
>>> bool(re.fullmatch(r'The.*Floss\.', x))
True
>>> bool(re.match(r'Mill', x))
False
>>> bool(re.search(r'Mill', x))
True
```
Retrieving Matched Text

- Match objects also carry information about what has been matched. The `.group()` method allows you to retrieve it.

```python
>>> x = "This string contains 35 characters."
>>> mat = re.search(r'\d+', x)
>>> mat.group()
'35'
```

- Furthermore, if there are parenthesized expressions in the pattern, you can retrieve them as well.

```python
>>> x = "There were 12 pence in a shilling and 20 shillings in a pound."
>>> mat = re.search(r'(\d+).*(\d+)', x)
>>> mat.group()  # Same as mat.group()
'12 pence in a shilling and 20'
>>> mat.group(1)
'12'
>>> mat.group(2)
'20'
>>> mat.groups()  # All parenthesized groups
('12', '20')
```
Finding All Matches

- Finally, we can sequence through all possible matches in a string:

```python
>>> x = "1/2, 3/6, apple, 15, goat, -26/2"
>>> for mat in re.finditer(r"(-?d+)/?\d+", x):
...    if mat.group(2) is None:
...        print(mat.group())
...    else:
...        print(f"{mat.group(1)} over {mat.group(3)}")
1 over 2
3 over 6
15
-26 over 2
```
Substitution

• The `re.sub` method substitutes for all matches to a pattern.

```python
>>> re.sub(r'\s+', '-', "Replace my whitespace with \ndashes")
'Replace-my-whitespace-with-dashes'
>>> re.sub(r'\s+', '', "Squeeze out blanks")
'Squeezeoutblanks'
```

• Furthermore, in the replacement string, you can use \1, \2, etc., to indicate you want the replacement to be one of the groups from the match:

```python
>>> re.sub(r'(\S+)<(\S+)', r'\2>\1', "I think that x<10 and y<0")
'I think that 10>x and 0>y'
```

• The replacement value can even be a function that is applied to each match:

```python
>>> re.sub(r'\d+', lambda x: str(int(x.group()) * 2), "1, 2, 3, 4, 5")
'2, 4, 6, 8, 10'
```
Resolving Ambiguity

- Classical regular expressions can match a given string in more than one way.
- Especially when there are parenthesized groups, this can lead to ambiguity:

```python
>>> mat = re.match(r'wind|window', 'window')
>>> mat.group()
# Is this 'wind' or 'window'?
>>> mat = re.match(r'window|wind', 'window')
>>> mat.group()  # Is this 'wind' or 'window'?
# Is this 'wind' or 'window'?
>>> mat = re.match(r'(wind|window)(.*)shade', 'window shade')
>>> mat.groups()
# ?
>>> mat = re.match(r'(window|wind)(.*)shade', 'window shade')
>>> mat.groups()
# ?
```

Python resolves these particular ambiguities in favor of the first option.
Resolving Ambiguity

- Classical regular expressions can match a given string in more than one way.

- Especially when there are parenthesized groups, this can lead to ambiguity:

```python
>>> mat = re.match(r'wind|window', 'window')
>>> mat.group()
'wind'
>>> mat = re.match(r'window|wind', 'window')
>>> mat.group()  # Is this 'wind' or 'window'?  # Is this 'wind' or 'window'?
>>> mat = re.match(r'(wind|window)(.*)shade', 'window shade')
>>> mat.groups()  # ?
>>> mat = re.match(r'(window|wind)(.*)shade', 'window shade')
>>> mat.groups()  # ?
```

Python resolves these particular ambiguities in favor of the first option.
Resolving Ambiguity

- Classical regular expressions can match a given string in more than one way.

- Especially when there are parenthesized groups, this can lead to ambiguity:

  ```python
  >>> mat = re.match(r'wind|window', 'window')
  >>> mat.group()
  'wind'
  >>> mat = re.match(r'window|wind', 'window')
  >>> mat.group()  # Is this 'wind' or 'window'?
  'window'
  >>> mat = re.match(r'(wind|window)(.*)shade', 'window shade')
  >>> mat.groups()
  # ?
  >>> mat = re.match(r'(window|wind)(.*)shade', 'window shade')
  >>> mat.groups()
  # ?
  ```

  Python resolves these particular ambiguities in favor of the first option.
Resolving Ambiguity

• Classical regular expressions can match a given string in more than one way.

• Especially when there are parenthesized groups, this can lead to ambiguity:

```python
>>> mat = re.match(r'wind|window', 'window')
>>> mat.group()
'wind'
>>> mat = re.match(r'window|wind', 'window')
>>> mat.group()  # Is this 'wind' or 'window'?
'window'
>>> mat = re.match(r'(wind|window)(.*)shade', 'window shade')
>>> mat.groups()
('wind', 'ow ')
>>> mat = re.match(r'(window|wind)(.*)shade', 'window shade')
>>> mat.groups()
('window', ' ')
```

Python resolves these particular ambiguities in favor of the first option.
Resolving Ambiguity (II)

• Likewise, there is ambiguity with '*', '+', and '?':

```python
>>> mat = re.match(r'(x*)(.*)', 'xxx')
>>> mat.groups()

>>> mat = re.match(r'(x+)(.*)', 'xxx')
>>> mat.groups()

>>> mat = re.match(r'(x?)(.*)', 'xxx')
>>> mat.groups()

>>> mat = re.match(r'(.*)/(.+)', '12/10/2020')
>>> mat.groups()

• That is, Python chooses to match **greedily**, matching the pattern left-to-right and, when given a choice, matching as much as possible while still allowing the rest of the pattern to match.
Resolving Ambiguity (II)

- Likewise, there is ambiguity with `*`, `+`, and `?`:

  ```python
  >>> mat = re.match(r'(x*)(.*)', 'xxx')
  >>> mat.groups()
  ('xxx', '')
  >>> mat = re.match(r'(x+)(.*)', 'xxx')
  >>> mat.groups()
  ('xxx', '')
  >>> mat = re.match(r'(x?)(.*)', 'xxx')
  >>> mat.groups()
  ('x', 'xx')
  >>> mat = re.match(r'(\.*)/(\.+)', '12/10/2020')
  >>> mat.groups()
  ?
  ```

- That is, Python chooses to match **greedily**, matching the pattern left-to-right and, when given a choice, matching as much as possible while still allowing the rest of the pattern to match.
Likewise, there is ambiguity with ‘*’, ‘+’, and ‘?’:

```python
>>> mat = re.match(r'(x*)(.*)', 'xxx')
>>> mat.groups()
('xxx', '')

>>> mat = re.match(r'(x+)(.*)', 'xxx')
>>> mat.groups()
('xxx', '')

>>> mat = re.match(r'(x?)(.*)', 'xxx')
>>> mat.groups()
('x', 'xx')

>>> mat = re.match(r'(.*)/(.+)', '12/10/2020')
>>> mat.groups()
('12/10', '2020')
```

That is, Python chooses to match *greedily*, matching the pattern left-to-right and, when given a choice, matching as much as possible while still allowing the rest of the pattern to match.

In the last example, the ( .*) doesn’t match the whole string, because then the second group couldn’t match.
Resolving Ambiguity: Laziness

- Sometimes, you don’t want to match as much as possible.
- The *lazy operators* `*?`, `+?`, and `??` match only as much as necessary for the whole pattern to match.

  ```python
  >>> mat = re.match(r'(.*)(\d*)', 'I have 5 dollars')
  >>> mat.groups()
  ('I have 5 dollars', '')
  >>> mat = re.match(r'(.*?)(\d+)', 'I have 5 dollars')
  >>> mat.groups()
  ('I have ', '5')
  >>> mat = re.match(r'(.*?)(\d*)', 'I have 5 dollars')
  >>> mat.groups()
  ('', '')
  ```

- Finally, the ambiguities introduced by `*`, `+`, `?`, and `|` don’t matter if all you care about is whether there is a match.
Your Turn

- Match a hexadecimal number in Python (starts with 0x).
- Match a list of words separated by commas and whitespace (such as "cat, dog, gnu, zebra").
- Match text in parentheses.
- Match text in parentheses that are not nested.