Regular expressions
Class outline:

- Declarative languages
- Regular expression syntax
- Regular expressions in Python
Declarative languages
Declarative programming

In **imperative** languages:

- A "program" is a description of computational processes
- The interpreter carries out execution/evaluation rules

In **declarative** languages:

- A "program" is a description of the desired result
- The interpreter figures out how to generate the result
- Examples:
  - Regular expressions: `Good (?:morning|evening)`
  - Backus-Naur Form:
    ```
    ?calc_expr: NUMBER | calc_op
    calc_op: "(" OPERATOR calc_expr* ")"
    OPERATOR: "+" | "-" | "*" | "/"
    ```
Domain-specific languages

Many declarative languages are **domain-specific**: they are designed to tackle problems in a particular domain, instead of being general purpose multi-domain programming languages.

<table>
<thead>
<tr>
<th>Language</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular expressions</td>
<td>Pattern-matching strings</td>
</tr>
<tr>
<td>Backus-Naur Form</td>
<td>Parsing strings into parse trees</td>
</tr>
<tr>
<td>SQL</td>
<td>Querying and modifying database tables</td>
</tr>
<tr>
<td>HTML</td>
<td>Describing the semantic structure of webpage content</td>
</tr>
<tr>
<td>CSS</td>
<td>Styling webpages based on selectors</td>
</tr>
<tr>
<td>Prolog</td>
<td>Describes and queries logical relations</td>
</tr>
</tbody>
</table>
Regular expressions
Pattern matching

Pattern matching in strings is a common problem in computer programming.

An imperative approach:

```python
def is_email_address(str):
    parts = str.split('@')
    if len(parts) != 2:
        return False
    domain_parts = parts[1].split('.
    return len(domain_parts) >= 2 and len(domain_parts[-1]) == 3
```
Pattern matching

Pattern matching in strings is a common problem in computer programming.

An imperative approach:

```python
def is_email_address(str):
    parts = str.split('@')
    if len(parts) != 2:
        return False
    domain_parts = parts[1].split('.')
    return len(domain_parts) >= 2 and len(domain_parts[-1]) == 3
```

An equivalent regular expression:

```
(.+)@(.+)\.(.{3})
```

With regular expressions, a programmer can just describe the pattern using a common syntax, and a regular expression engine figures out how to do the pattern matching for them.
Matching exact strings

The following are special characters in regular expressions: \ ( ) [ ] { } + * ? | $ ^ .

To match an exact string that has no special characters, just use the string:

Berkeley, CA 94720

Fully matched by: Berkeley, CA 94720

But if the matched string contains special characters, they must be escaped using a backslash.

\(1\+3\)

Fully matched by: (1+3)
The . character matches any single character that is not a new line.

`.a.a.a`

Fully matched by: `banana`

It's typically better to match a more specific range of characters, however...
## Character classes

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
<th>Fully Matched By</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Denotes a character class. Matches characters in a set (including ranges of characters like [0-9]). Use ^[^] to match characters outside a set.</td>
<td>[top]</td>
<td>t</td>
</tr>
<tr>
<td>[]</td>
<td>Denotes a character class. Matches characters in a set (including ranges of characters like [0-9]). Use ^[^] to match characters outside a set.</td>
<td>[h-p]</td>
<td>j</td>
</tr>
<tr>
<td>.</td>
<td>Matches any character other than the newline character.</td>
<td>1.</td>
<td>1?</td>
</tr>
<tr>
<td>\d</td>
<td>Matches any digit character. Equivalent to [0-9]. \d matches the inverse (all non-digit characters).</td>
<td>\d\d</td>
<td>12</td>
</tr>
<tr>
<td>\w</td>
<td>Matches any word character. Equivalent to [A-Za-z0-9]. \w matches the inverse.</td>
<td>\d\w</td>
<td>4Z</td>
</tr>
<tr>
<td>\s</td>
<td>Matches any whitespace character: spaces, tabs, or line breaks. \s matches the</td>
<td>\d\s\w</td>
<td>9 a</td>
</tr>
</tbody>
</table>
# Quantifiers

These indicate how many of a character/character class to match.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
<th>Fully Matched By</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Matches 0 or more of the previous pattern.</td>
<td>a*</td>
<td>aaa</td>
</tr>
<tr>
<td>+</td>
<td>Matches 1 or more of the previous pattern.</td>
<td>lo+l</td>
<td>lool</td>
</tr>
<tr>
<td>?</td>
<td>Matches 0 or 1 of the previous pattern.</td>
<td>lo?l</td>
<td>lol</td>
</tr>
<tr>
<td>{}</td>
<td>Used like {Min, Max}. Matches a quantity between Min and Max of the previous pattern.</td>
<td>a{2}</td>
<td>aa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a{2,}</td>
<td>aaaaaa</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a{2,4}</td>
<td>aaa</td>
</tr>
</tbody>
</table>
# Combining patterns

Patterns $P_1$ and $P_2$ can be combined in various ways.

<table>
<thead>
<tr>
<th>Combination</th>
<th>Description</th>
<th>Example</th>
<th>Fully Matched By</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1P_2$</td>
<td>A match for $P_1$ followed immediately by one for $P_2$.</td>
<td>ab[.,]</td>
<td>ab,</td>
</tr>
<tr>
<td>$P_1</td>
<td>P_2$</td>
<td>Matches anything that either $P_1$ or $P_2$ does.</td>
<td>\d+</td>
</tr>
<tr>
<td>$(P_1)$</td>
<td>Matches whatever $P_1$ does. Parentheses group, just as in arithmetic expressions.</td>
<td>(&lt;3)+</td>
<td>&lt;3&lt;3&lt;3</td>
</tr>
</tbody>
</table>
# Anchors

These don't match an actual character, they indicate the position where the surrounding pattern should be found.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Example</th>
<th>What parts match?</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Matches the beginning of a string.</td>
<td>^aw+</td>
<td>aww aww</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of a string.</td>
<td>\w+y$</td>
<td>stay stay</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a word boundary, the beginning or end of a word.</td>
<td>\w+e\b</td>
<td>broken bridge team</td>
</tr>
</tbody>
</table>
Regular expressions in Python
Support for regular expressions

Regular expressions are supported natively in many languages and tools.

Languages: Perl, ECMAScript, Java, Python, ..

Tools: Excel/Google Spreadsheets, SQL, BigQuery, VSCode, grep, ...
Raw strings

In normal Python strings, a backslash indicates an escape sequence, like `\n` for new line or `\b` for bell.

```python
g>>> print("I have\na newline in me.")
I have
a newline in me
```

But backslash has a special meaning in regular expressions. To make it easy to write regular expressions in Python strings, use raw strings by prefixing the string with an `r`:

```python
epattern = r"\b[ab]+\b"
```
## The re module

The **re module** provides many helpful functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>re.search(pattern, string)</code></td>
<td>returns a <strong>Match</strong> object representing the first occurrence of pattern within string</td>
</tr>
<tr>
<td><code>re.fullmatch(pattern, string)</code></td>
<td>returns a <strong>Match</strong> object, requiring that pattern matches the entirety of string</td>
</tr>
<tr>
<td><code>re.match(pattern, string)</code></td>
<td>returns a <strong>Match</strong> object, requiring that string starts with a substring that matches pattern</td>
</tr>
<tr>
<td><code>re.findall(pattern, string)</code></td>
<td>returns a list of strings representing all matches of pattern within string, from left to right</td>
</tr>
<tr>
<td><code>re.sub(pattern, repl, string)</code></td>
<td>substitutes all matches of pattern within string with repl</td>
</tr>
</tbody>
</table>
Match objects

The functions `re.search`, `re.match`, 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`.

`re.search` requires that the pattern exists somewhere in the string:

```python
import re

re.search(r'-?\d+', '123 peeps')  # <re.Match object>
re.search(r'-?\d+', 'So many peeps')  # None
```
Match objects

The functions `re.search`, `re.match`, 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`.

`re.search` requires that the pattern exists somewhere in the string:

```python
import re

re.search(r'-\d+', '123 peeps')  # <re.Match object>
re.search(r'-\d+', 'So many peeps')  # None
```

Match objects are treated as true values, so you can use the result as a boolean:

```python
bool(re.search(r'-\d+', '123'))  # True
bool(re.search(r'-\d+', 'So many peeps'))  # False
```
Inspecting a match

`re.search` returns a `Match` object representing the first occurrence of pattern within string.

```python
title = "I Know Why the Caged Bird Sings"
re.search(r'Bird', title)  #
```

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

```python
x = "This string contains 35 characters."
mat = re.search(r'\d+', x)
mat.group(0)  # 35
```
Match groups

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```python
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'\d+\s[a-z]+\d+', x)
mat.group(0)
mat.group(1)
mat.group(2)
mat.groups()
```

It's also common to use parentheses in combination with quantifiers and other modifiers, however.
Match groups

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```python
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'(\d+) [a-z]+(\d+)', x)

mat.group(0)  # '12 pence in a shilling and 20'
mat.group(1)
mat.group(2)
mat.groups()
```

It's also common to use parentheses in combination with quantifiers and other modifiers, however.
Match groups

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```python
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'(\d+)\s([a-z]+)(\d+)', x)

mat.group(0)  # '12 pence in a shilling and 20'
mat.group(1)  # 12
mat.group(2)  # 'pence in a shilling and 20'
mat.groups()
```

It's also common to use parentheses in combination with quantifiers and other modifiers, however.
Match groups

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```python
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'(\d+) [a-z\s]+(\d+)', x)
```

```python
mat.group(0)  # '12 pence in a shilling and 20'
mat.group(1)  # 12
mat.group(2)  # 20
mat.groups()
```

It's also common to use parentheses in combination with quantifiers and other modifiers, however.
Match groups

If there are parentheses in a patterns, each of the parenthesized groups will become groups in the match object.

```python
x = "There were 12 pence in a shilling and 20 shillings in a pound."
mat = re.search(r'\d+[a-z]+\d+', x)
```

```python
mat.group(0)    # '12 pence in a shilling and 20'
mat.group(1)    # 12
mat.group(2)    # 20
mat.groups()    # (12, 20)
```

It's also common to use parentheses in combination with quantifiers and other modifiers, however.
Exercises
Name That Pattern! #1

[A-Za-z] {3}

Fully matched by: ?

- What's a valid input?
- What's an invalid input?
Name That Pattern! #1

[A-Za-z] {3}

Fully matched by: ?

- What's a valid input? AUS, aus
- What's an invalid input? australia, au
Name That Pattern! #2

\d{4} - \d{2} - \d{2}

Fully matched by: ?

- What's a valid input?
- What's an invalid input?
Name That Pattern! #2

\d{4} - \d{2} - \d{2}

Fully matched by: 2020-03-13

- What's a valid input? 2020-03-13
- What's an invalid input? 2020/03/13, 03-13-2020
Name That Pattern! #3

\[ [a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]\{2,\}$ \]

Fully matched by: ?

- What's a valid input?
- What's an invalid input?
Name That Pattern! #3

\[(a-z0-9._%+-)+@[a-z0-9.-]+\.[a-z]\{2,\}\$

Fully matched by: ?

- What's a valid input? someone@someplace.org
- What's an invalid input? someone@mod%cloth.co
Exercise: Stocks

Make a regular expression to match any tweet talking about GME stock.

```python
import re

def match_gme(tweet):
    """
    >>> match_gme('GME')
    True
    >>> match_gme('yooo buy GME right now!')
    True
    >>> match_gme('#HUGME')
    False
    >>> match_gme('#HUGMEHARDER')
    False
    """
    return bool(re.search(______, tweet))
```
Tips

• When learning, use sites like regexr.com
• Get used to referencing a regular expressions cheat sheet
⚠️ **A word of caution ⚠️**

Regular expressions can be very useful. However:

- **Very long regular expressions** can be difficult for other programmers to read and modify.  
  See also: Write-only

- Since regular expressions are declarative, it's not always clear how efficiently they'll be processed. Some processing can be so time-consuming, it can **take down a server**.

- Regular expressions can't parse everything! **Don't write an HTML parser with regular expressions.**