Regular expressions

Class outline:

- Declarative languages
- Regular expression syntax
- Regular expressions in Python
- Ambiguous regular expressions

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: "+" | "-" | "*" | "/"
    ```
  - SQL: `select max(longitude) from cities where longitude >= 115`

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.

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

Regular expressions

Pattern matching

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

An imperative approach:

```
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:

```
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
```

Matches: Berkeley, CA 94720

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

```
\(1\+3\)
```

Matches: (1+3)

The dot

The `.` character matches any single character that is not a new line.

```
.a.a.a
```

Matches: banana

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

Character classes

Pattern	Description	Example	Matches:
`[]`	Denotes a character class. Matches characters in a set (including ranges of characters like `0-9`). Use `[^]` to match characters outside a set.	`[top]`	t
`.`	Matches any character other than the newline character.	`1.`	1?
`\d`	Matches any digit character. Equivalent to `[0-9]`. `\D` is the complement and refers to all non-digit characters.	`\d\d`	12
`\w`	Matches any word character. Equivalent to `[A-Za-z0-9_]`. `\W` is the complement.	`\d\w`	4Z
`\s`	Matches any whitespace character: spaces, tabs, or line	`\d\s\w`	9 a

Quantifiers

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

Pattern	Description	Example	Matches:
`*`	Matches 0 or more of the previous pattern.	`a*`	aaa
`+`	Matches 1 or more of the previous pattern.	`lo+l`	lool
`?`	Matches 0 or 1 of the previous pattern.	`lo?l`	lol
`{}`	Used like `{Min, Max}`. Matches a quantity between Min and Max of the previous pattern.	`a{2,4}`	aaaa

Anchors

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

Pattern	Description	Example	Matches:
`^`	Matches the beginning of a string.	`^aw+`	aww
`$`	Matches the end of a string.	`\w+y$`	stay
`\b`	Matches a word boundary, the beginning or end of a word.	`\w+e\b`	bridge

Combining patterns

Patterns `P1` and `P2` can be combined in various ways.

Combination	Description	Example	Matches:
`P1P2`	A match for P1 followed immediately by one for P2.	`ab[.,]`	ab,
`P1|P2`	Matches anything that either P1 or P2 does.	`\d+|Inf`	Inf
`(P1)`	Matches whatever P1 does. Parentheses group, just as in arithmetic expressions.	`(<3)+`	<3<3<3

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.

```
>>> 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`:

```
pattern = r"\b[ab]+\b"
```

The re module

The re module provides many helpful functions.

Function	Description
`re.search(pattern, string)`	returns a match object representing the first occurrence of pattern within string
`re.fullmatch(pattern, string)`	returns a match object, requiring that pattern matches the entirety of string
`re.match(pattern, string)`	returns a match object, requiring that string starts with a substring that matches pattern
`re.findall(pattern, string)`	returns a list of strings representing all matches of pattern within string, from left to right
`re.sub(pattern, repl, string)`	substitutes all matches of pattern within string with repl

Match objects

The functions `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`.

`re.fullmatch` requires that the pattern matches the entirety of the string:

```
import re

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

Match objects

The functions `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`.

`re.fullmatch` requires that the pattern matches the entirety of the string:

```
import re

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

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

```
bool(re.fullmatch(r'-?\d+', '123'))        # True
bool(re.fullmatch(r'-?\d+', '123 peeps')) # False
```

Inspecting a match

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

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

Inspecting a match

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

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

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

```
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.

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

```
mat.group(0)
mat.group(1)
mat.group(2)
mat.groups()
```

Match groups

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

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

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

Match groups

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

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

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

Match groups

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

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

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

Match groups

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

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

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

Finding multiple matches

`re.findall()` returns a list of strings representing all matches of pattern within string, from left to right.

```
locations = "CA 91105, NY 13078, CA 94702"
re.findall(r'\d\d\d\d\d', locations)
# ['91105', '13078', '94702']
```

Resolving ambiguity

Ambiguous matches

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

```
mat = re.match(r'wind|window', 'window')
mat.group()

mat = re.match(r'window|wind', 'window')
mat.group()

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.

Ambiguous matches

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

```
mat = re.match(r'wind|window', 'window')
mat.group()  # 'wind'

mat = re.match(r'window|wind', 'window')
mat.group()

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.

Ambiguous matches

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

```
mat = re.match(r'wind|window', 'window')
mat.group()  # 'wind'

mat = re.match(r'window|wind', 'window')
mat.group() # '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.

Ambiguous matches

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

```
mat = re.match(r'wind|window', 'window')
mat.group()  # 'wind'

mat = re.match(r'window|wind', 'window')
mat.group() # '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()
```

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

Ambiguous matches

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

```
mat = re.match(r'wind|window', 'window')
mat.group()  # 'wind'

mat = re.match(r'window|wind', 'window')
mat.group() # '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.

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

```
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()
```

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.

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

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

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()
```

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.

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

```
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()

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

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.

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

```
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()
```

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.

Ambiguous quantifiers

Likewise, there is ambiguity with `*`, `+`, and `?`.

```
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')
```

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.

Lazy operators

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.

```
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() # ('', '')
```

The ambiguities introduced by `*`, `+`, `?`, and | don't matter if all you care about is whether there is a match!

⚠ 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.