RE Literals

Compiling Regular Expressions

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

- >>>
- >>> import re
- >>> p = re.compile('ab*')
- >>> p
- re.compile('ab*')

enum

- This module defines four enumeration classes that can be used to define unique sets of names and values

- >> from enum import Enum
- >>> class Color(Enum):
- ...   RED = 1
- ...   GREEN = 2
- ...   BLUE = 3
- ...

enum

- >>> print(Color.RED)
- Color.RED
- …while their repr has more information:
- >>> print(repr(Color.RED))
- <Color.RED: 1>
- The type of an enumeration member is the enumeration it belongs to:
- >>> type(Color.RED)
- <enum 'Color'>
- >>> isinstance(Color.GREEN, Color)
- True
- Enum members also have a property that contains just their item name:

- >>> print(Color.RED.name)
- RED

Enumeration

- Enumeration members are hashable, so they can be used in dictionaries and sets:

- >>>
- >>> apples = {}
- >>> apples[Color.RED] = 'red delicious'
- >>> apples[Color.GREEN] = 'granny smith'
- >>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
- True

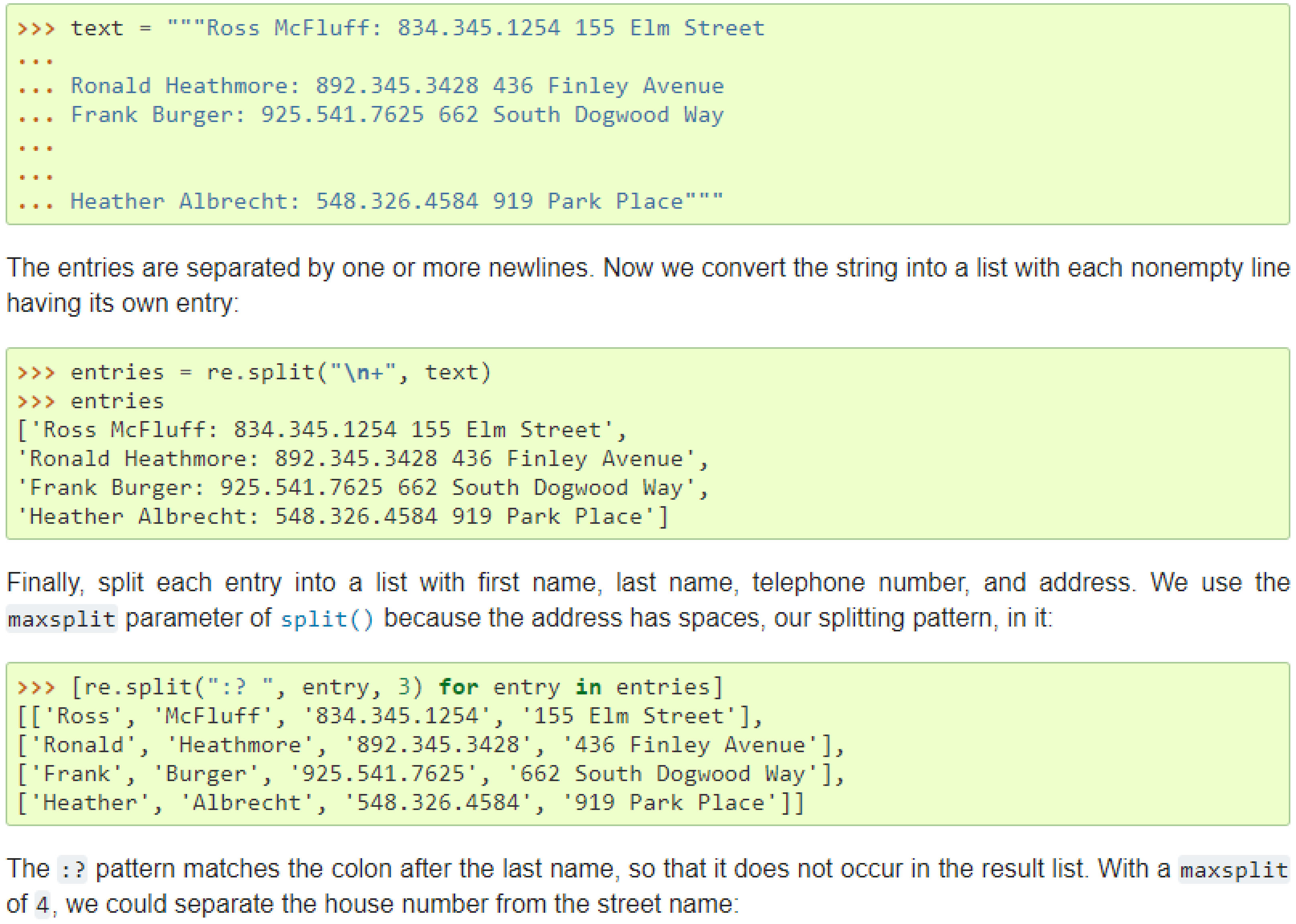
search() vs. match()

- Python offers two different primitive operations based on regular expressions: re.match() checks for a match only at the beginning of the string, while re.search() checks for a match anywhere in the string (this is what Perl does by default).
- For example:
- `>>>`
- `>>> re.match("c", "abcdef")    # No match`
- `>>> re.search("c", "abcdef")   # Match`
- `<re.Match object; span=(2, 3), match='c'>`

split()

- split() splits a string into a list delimited by the passed pattern. The method is invaluable for converting textual data into data structures that can be easily read and modified by Python as demonstrated in the following example that creates a phonebook.

Split



```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

The entries are separated by one or more newlines. Now we convert the string into a list with each nonempty line having its own entry:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

Finally, split each entry into a list with first name, last name, telephone number, and address. We use the `maxsplit` parameter of `split()` because the address has spaces, our splitting pattern, in it:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

The `:?` pattern matches the colon after the last name, so that it does not occur in the result list. With a `maxsplit` of `4`, we could separate the house number from the street name:

Finding all Adverbs

- findall() matches all occurrences of a pattern, not just the first one as search() does. For example, if a writer wanted to find all of the adverbs in some text, they might use findall() in the following manner:

- >>>
- >>> text = "He was carefully disguised but captured quickly by police."
- >>> re.findall(r"\w+ly", text)
- ['carefully', 'quickly']

Finding all Adverbs and their Positions

- If one wants more information about all matches of a pattern than the matched text, finditer() is useful as it provides match objects instead of strings. Continuing with the previous example, if a writer wanted to find all of the adverbs and their positions in some text, they would use finditer() in the following manner:
- >>> text = "He was carefully disguised but captured quickly by police."
- >>> for m in re.finditer(r"\w+ly", text):
- ...   print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
- 07-16: carefully
- 40-47: quickly

fullmatch

- If the whole string matches this regular expression, return a corresponding match object. Return None if the string does not match the pattern; note that this is different from a zero-length match.

- The optional pos and endpos parameters have the same meaning as for the search() method.

- >>>
- >>> pattern = re.compile("o[gh]")
- >>> pattern.fullmatch("dog")   # No match as "o" is not at the start of "dog".
- >>> pattern.fullmatch("ogre")   # No match as not the full string matches.
- >>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
- <re.Match object; span=(1, 3), match='og'>

re.escape(pattern)

- re.escape(pattern)
- Escape special characters in pattern. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

- >>>
- >>> print(re.escape('http://www.python.org'))
- http://www\.python\.org

Exam 2:

- >>> legal_chars = "!#$%&'*+-.^_`|~:"
- >>> print(re.escape(legal_chars))
- !\#\$%\&'*\+\-\.\^_`\|\~:

search

- >>> pattern = re.compile('bar')
- >>> pattern.search('foobar')
- <_sre.SRE_Match object at 0x103c9a578>
- >>> _.start()
- 3

Sub and subn

- import re
- text = "Python for beginner is a very cool website"
- pattern = re.sub("cool", "good", text)
- print text2

- Subn: Perform the same operation as sub(), but return a tuple (new_string, number_of_subs_made).