Lecture 26: Regular Expressions

Regular Expressions

Regular expressions are a small programming language over strings

- **Regex** or **regexp** are not unique to Python
- They let us to succinctly and compactly represent *classes* of strings

In this class we will use them to scan chunks of text and *match* strings.

Regular Expressions

Python supports regular expressions in the `re` module.

```
>>> import re
```

A basic text string can be a regex that performs exact matching:

```
>>> re.search("step", "I never half step cause I'm not a half stepper")
<_sre.SRE_Match object; span=(13, 17), match='step'>
>>> re.search("stop","I never half step cause I'm not a half stepper.")
>>>
```

Regular Expressions

Python supports regular expressions in the `re` module.

```
>>> import re
```

A basic text string can be a regex that performs exact matching:

```
>>> re.search("step", "I never half step cause I'm not a half stepper")
<_sre.SRE_Match object; span=(13, 17), match='step'>
>>> re.search("stop","I never half step cause I'm not a half stepper.")
>>>
```

`re.search` scans the whole string for the first match and returns an `SRE_Match` or `None`

Python Regex Functions

Python provides four primary methods to search text for patterns expressed as regular expressions.

`match` checks if the regular expression matches at the *beginning* of the text;

`search` finds the first matching location of a pattern in a text;

`findall` finds all the locations of the pattern within the text and returns them as a list;

`finditer` finds all the locations of of the pattern within the text and returns an iterator.

Regular Expressions

Special characters have their own meaning, and they make the language powerful:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Special characters have their own meaning, and they make the language powerful:

```
. ^ $ * + ? { } [ ] \ | ( )
```

To use one of these characters literally, we must escape it

```
>>> re.search("u \+ m", "I know my calculus. It says you + me = us")
<_sre.SRE_Match object; span=(30, 35), match='u + m'>
```

But we often use the special characters *as* special characters:

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
```

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
['hop', 'pop']
```

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
['hop', 'pop']
```

- A '*' matches 0 or more of a thing.

```
>>> re.findall("be*", "beets, bears, battlestar galactica")
```

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
['hop', 'pop']
```

- A '*' matches 0 or more of a thing.

```
>>> re.findall("be*", "beets, bears, battlestar galactica")
['bee', 'be', 'b']
```

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
['hop', 'pop']
```

- A '*' matches 0 or more of a thing.

```
>>> re.findall("be*", "beets, bears, battlestar galactica")
['bee', 'be', 'b']
```

- A '+' matches 1 or more of a thing.

```
>>> re.findall("be+", "beets, bears, battlestar galactica")
```

But we often use the special characters *as* special characters:

- A '.' matches any single character.

```
>>> re.findall(".op", "hop on pop")
['hop', 'pop']
```

- A '*' matches 0 or more of a thing.

```
>>> re.findall("be*", "beets, bears, battlestar galactica")
['bee', 'be', 'b']
```

- A '+' matches 1 or more of a thing.

```
>>> re.findall("be+", "beets, bears, battlestar galactica")
['bee', 'be']
```

Regular Expressions

- Brackets '[ ]' match a character class
  - '[abc]' would match an 'a' or a 'b' or a 'c'
  - '[0-9]' would match any single decimal digit
  - '[A-Za-z]' would match any single letter, capital or lowercase

```
>>> re.findall("[1-3a-c]", "ABC, it's easy as 123")
```

Regular Expressions

- Brackets '[ ]' match a character class
  - '[abc]' would match an 'a' or a 'b' or a 'c'
  - '[0-9]' would match any single decimal digit
  - '[A-Za-z]' would match any single letter, capital or lowercase

```
>>> re.findall("[1-3a-c]", "ABC, it's easy as 123")

['a', 'a', '1', '2', '3']
```

Regular Expressions

- Brackets '[ ]' match a character class
  - '[abc]' would match an 'a' or a 'b' or a 'c'
  - '[0-9]' would match any single decimal digit
  - '[A-Za-z]' would match any single letter, capital or lowercase

```
>>> re.findall("[1-3a-c]", "ABC, it's easy as 123")

['a', 'a', '1', '2', '3']

>>> re.findall("[1-3A-C]+", "ABC, it's easy as 123")
```

Regular Expressions

- Brackets '[ ]' match a character class
  - '[abc]' would match an 'a' or a 'b' or a 'c'
  - '[0-9]' would match any single decimal digit
  - '[A-Za-z]' would match any single letter, capital or lowercase

```
>>> re.findall("[1-3a-c]", "ABC, it's easy as 123")

['a', 'a', '1', '2', '3']

>>> re.findall("[1-3A-C]+", "ABC, it's easy as 123")

['ABC', '123']
```

```
>>> re.findall("              ", "the rain in spain stays mainly on the plain")
['rain', 'spain', 'plain']
```

```
>>> re.findall("              ", "the rain in spain stays mainly on the plain")
['rain', 'spain', 'plain']

>>> re.findall("[sp]*[rpl]ain", "the rain in spain stays mainly on the plain")
```

Other Special Characters

`?` means the previous character in the regular expression is optional

- `0?01` matches `001` and `01`
- `?` following a `*` (or a `+`) means be minimally greedy in the match.

`{m}` means match exact `m` copies of the previous character.

`{m,n}` means match between `m` and `n` characters.

- For example, a/{1,3}b will match a/b, a//b, and a///b. It won?t match ab, which has no slashes, or a////b, which has four.
- The final `n` may be omitted (but the comma must remain) to give a lower bound on the number of characters
- One can also append the `?` to this (e.g., `{3,5}?`) to minimally match the requirement.

`^` is used to preface part of a pattern that only matches at the start of the text.

`$` is used to indicate that a pattern should reach the end of the text.

`( )` are used to extract portions of a matched pattern using `SRE_Match.group(i)`

Regex Groups

```
www = ["http://math.williams.edu/best-jobs-2015/",
       "http://www.williams.edu/registrar",
       "http://magazine.williams.edu/2015/spring/study/the-body-as-book/"]
```

With groups, we can to isolate text inside a larger matched pattern

- Groups are defined by the ( ) special characters

```
>>> [re.match("http://(.*?)\.(.*?)/",w).group(0) for w in www]
['http://math.williams.edu/',
 'http://www.williams.edu/',
 'http://magazine.williams.edu/']
>>> [re.match("http://(.*?)\.(.*?)/",w).group(1) for w in www]
['math', 'www', 'magazine']
>>> [re.match("http://(.*?)\.(.*?)/",w).group(2) for w in www]
['williams.edu', 'williams.edu', 'williams.edu']
```

Hexadecimal Colors

Write a regular expression to match a hexadecimal color value in a piece of text. A hexadecimal color value is a 6 character sequence where each character is a hexadecimal digit (i.e. between 0 and f) preceded by an optional #. For example `#ff34d5` is valid but `#h56732` is not. Make sure to group the actual hex number for ease-of-use.

Hexadecimal Colors

Write a regular expression to match a hexadecimal color value in a piece of text. A hexadecimal color value is a 6 character sequence where each character is a hexadecimal digit (i.e. between 0 and f) preceded by an optional #. For example `#ff34d5` is valid but `#h56732` is not. Make sure to group the actual hex number for ease-of-use.

```
#?([0-9A-Fa-f]{6})
```

IP Addresses

IP addresses are strings of four numbers, delimited by a period, where each number is in the range $[0, 255]$. For example, the IP address of this computer is 137.165.206.66. The IP address for the Google Domain Name Server is 8.8.8.8, which can also be written as 8.08.008.8. Write a regular expression to check if some text is exactly an IP address. That is, do IP address validation.

IP Addresses

IP addresses are strings of four numbers, delimited by a period, where each number is in the range $[0, 255]$. For example, the IP address of this computer is 137.165.206.66. The IP address for the Google Domain Name Server is 8.8.8.8, which can also be written as 8.08.008.8. Write a regular expression to check if some text is exactly an IP address. That is, do IP address validation.

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IP Addresses

IP addresses are strings of four numbers, delimited by a period, where each number is in the range $[0, 255]$. For example, the IP address of this computer is 137.165.206.66. The IP address for the Google Domain Name Server is 8.8.8.8, which can also be written as 8.08.008.8. Write a regular expression to check if some text is exactly an IP address. That is, do IP address validation.

```
^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

And here's a way to programmatically create the regular expression:

```
ips = []
for i in range(256):
    if (i < 10):
        ips.append(str(i).zfill(2))
    if (i < 100):
        ips.append(str(i).zfill(3))
    ips.append(str(i))

regexips ="^(({0})\\.){{3}}({0})$".format("|".join(num for num in ips))
```

Email Addresses

Write a regular expression to check whether some given text is a *valid* email address. A valid email address may contain the characters `.`, `%`, `+`, and `-`. Suppose, incorrectly, that all email addresses must end with a a 2-4 character string.

Email Addresses

Write a regular expression to check whether some given text is a *valid* email address. A valid email address may contain the characters `.`, `%`, `+`, and `-`. Suppose, incorrectly, that all email addresses must end with a a 2-4 character string.

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$
```