

# **Analyzing Text With Regular Expressions**

# Dealing with Text

How have you dealt with text in Python so far?

# Dealing with Text

How have you dealt with text in Python so far?

- Writing to CSV's
- Test equality
- Look for HTML snippets
- Use class-type variables in regression
- Passing arguments to functions
- Many other ways...

# Dealing with Text

In the past, as we have dealt with text, we have needed to meet EXACT conditions in order for our programs to be able to function:

```
>>> "mystring" == "MyString"  
False  
>>> "mystring" == "MyString".lower()  
True
```

If the exact condition is not met, then we cannot apply conditions based on this equality.

# Becoming More Flexible

Sometimes I survey my students before tests, and ask what score they expect to get (from 0 to 100%). How could I extract their number responses?

1. "I think I'll get a 50."
2. "47%"
3. "If I get a 60 I'm good."
4. "Don't know. Maybe 85."

Is there a good way to write Python code to catch all of these numbers?

# Becoming More Flexible

What if I have a form that asks for a phone number?  
How could I recognize valid US phone numbers  
among the following:

1. 425-389-1180

2. 3365-1328

3. 1-402-554-3303

4. 644-1428

5. 32-1845-9865

6. 846-16-9975

# Regular Expression

Regular Expression is a set of expressions through which we can describe the language that we are searching for. It is designed to find patterns in text, allowing for broad application of text conditions to data.

To use Regular Expression in Python,

```
import re
```

We will work through examples in the Python Console (either in Spyder or a terminal)

# Regular Expression

```
mystring = "I think I'll get a 50."
```

Let's find the expected score in this response!

```
re.search(r'[0123456789]', mystring)
```

- `r''` denotes a "raw" string, and permits escape characters. We will put our regex code into raw strings



# Regular Expression

```
mystring = "I think I'll get a 50."
```

Let's find the expected score in this response!

```
re.search(r'[0123456789]', mystring)
```

- `[]` denotes a "character class", or all possible values of a character that we are looking for
- In this case, we declare that we are looking for a character taking the value of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9

# Regular Expression

```
mystring = "I think I'll get a 50."  
re.search(r'[0123456789]', mystring)
```

What is our output?

```
<_sre.SRE_Match object; span=(19, 20), match='5'>
```

This is a "match object" generated by our search. It tells us 1) that we found a match, 2) the position of the match in the string, and 3) the string of the match itself

# Regular Expression

```
mystring = "I think I'll get a 50."  
re.search(r'[0123456789]', mystring)
```

What is our output?

```
<_sre.SRE_Match object; span=(19, 20), match='5'>
```

Did our student report that they would score a 5%?  
NO!

What is wrong?

# Regular Expression

```
mystring = "I think I'll get a 50."  
re.search(r'[0123456789]', mystring)
```

This search looks for a **single character** that has a number value. We need more than one character:

```
re.search(r'[0123456789][0123456789]', mystring)
```

This search returns the following:

```
<_sre.SRE_Match object; span=(19, 21), match='50'>
```

# Regular Expression

```
mystring = "I think I'll get a 50."  
re.search(r'[0123456789][0123456789]', mystring)
```

```
<_sre.SRE_Match object; span=(19, 21), match='50'>
```

Great! We match 50. But what if the student **did** respond with 5%? Or what if they thought they would get 100%? Try our search on new responses:

- "I think I'll get a 5."
- "I think I'll get a 100."

# Regular Expression

```
re.search(r'[0123456789][0123456789]', mystring)
```

- "I think I'll get a 5."
  - We get no result!
- "I think I'll get a 100."
  - Even worse, we get a BAD result!

What are all the possible numbers that are appropriate for our problem?

# Repeating Values

```
re.search(r'[0-9]*', mystring)
```

This new search will look for any number of consecutive number characters between 0 and 9 (all digits)

```
re.search(r'[0-9]+', mystring)
```

This search will look for **1 or more** digits between 0 and 9

# Repeating Values

- `*` is shorthand for 0 or more of a character/character class
- `+` is shorthand for 1 or more of a character/character class
- `?` is shorthand for 0 or 1 (like saying "maybe")
- `{x, y}` is shorthand for a character that is repeated no less than  $x$  times, and no more than  $y$  times

For percentages, we might want something like

```
re.search(r'[0-9]{1,3}', mystring)
```



# Uncertainty

```
mystring = "I think I'll get a 500."  
re.search(r'[0-9]{1,3}', mystring)
```

If students provide bad percentages, our search will still return them.

- "I think I'll get a 500."
- "I think I'll get a 5000."
- "I think I'll get a 12976589."

What do we get back in these cases?

# Uncertainty

We can be more stringent in our expectations:

- A percentage with 3 digits can only begin with a 1 or 0! That would then be followed by two digits between 0 and 9, but only if the first digit was not 1.

```
re.search(r'(100|[1-9][0-9]|[0-9])', mystring)
```

# Uncertainty

```
re.search(r'(100|[1-9][0-9]|[0-9])', mystring)
```

Here, we use a new symbol: **|** (call it the "pipe")

- Pipes are used to denote options

What we are saying is that our result should be **100** or a number with the first digit between 1 and 9 and the second digit between 0 and 9, or a single digit between 0 and 9.

# Uncertainty

```
re.search(r'(100|[1-9][0-9]|[0-9])', mystring)
```

Are there still problems here?

# Uncertainty

```
re.search(r'(100|[1-9][0-9]|[0-9])', mystring)
```

Are there still problems here?

Yes! We need to figure out why

1. 5000 still matches and returns 50
2. 500 does the same
3. 12976589 returns 12

# Establishing Boundaries

We need symbols that represent the start or end of words:

- `\b` represents a word boundary (either at the start or end of a word)
- `^` represents the start of a string
- `$` represents the end of a string

```
re.search(r'(100|[1-9][0-9]|[0-9])', mystring)
```

What do we need to add to our expression?

# Establishing Boundaries

We can solve our number problem with word boundaries in this case:

```
re.search(r'\b(100|[1-9][0-9]|[0-9])\b', mystring)
```

# Phone Numbers

How can we solve our phone number problem?

Take 5 minutes and try with your neighbors.

1. 425-389-1180

2. 3365-1328

3. 1-402-554-3303

4. 644-1428

5. 32-1845-9865

6. 846-16-9975



# Phone Numbers

How can we solve our phone number problem?

My solution:

```
re.search(r'\b((1-)?\d{3}-)?(\d{3}-\d{4})\b', mystring)
```

# Some Additional Shorthand

- `\w` - denotes any alphanumeric character, or an underscore (`[a-zA-Z0-9_]`)
- `\d` - denotes any numeric character (`[0-9]`)
- `\s` - denotes any whitespace character (`[ \t\n\r\f\v]`)
- `\W` - the inverse of `\w`
- `\D` - the inverse of `\d`
- `\S` - the inverse of `\s`

# City, State Combinations

What if we want to find city, state abbreviation combinations (ie Miami, FL) from a text address?

1. "6708 Pine Street, Omaha, NE 68182"
2. "1600 Pennsylvania Ave NW, Washington, DC 20006"
3. "261 S 800 E\nSalt Lake City, UT 84102"

Take 5 minutes to try it out. What shorthand might help?

# City, State Combinations

What if we want to find city, state abbreviation combinations (ie Miami, FL) from a text address?

```
re.search(r'((\b\w+\b)+(\s)?){1,3}, ([A-Z]{2})', mystring)
```

1. "6708 Pine Street, Omaha, NE 68182"
2. "1600 Pennsylvania Ave NW, Washington, DC 20006"
3. "261 S 800 E\nSalt Lake City, UT 84102"

This code assumes that a city name will not consist of more than 3 words. (Dangerous assumption)

# City, State Combinations

What if we want to find city, state abbreviation combinations (ie Miami, FL) from a text address?

```
myexp = r'(?:(?:\n|, )(.*)?(?:, )([A-Z]{2}))'
```

This will get us what we want without the constraints on city name

What is going on, though?

# Groups

What is this statement doing?

```
myexp = r'(?:(?:\n|,)(.*)((?:,)([A-Z]{2})))'
```

( and ) allow us to denote **groups** in our expression

- We have one overarching group (the outer group)
- We have a series of inner groups, breaking our code into small segments

# Groups

What is this statement doing?

```
myexp = r'(?:(?:\n|,)(.*)((?:,)([A-Z]{2})))'
```

Within our groups, we have a new, unique symbol: `?:`

- When used inside a group, this indicates that we do not care to capture the group
- This is useful when we change how we use the `re` library

# Alternate Functionality

The `re` library offers several functions:

1. `re.search`: We can search as we have so far
2. `re.findall`: We can search for all matches in a string
3. `re.finditer`: We can generate an iterator to process all matches in a string (related to `findall`)
4. `re.split`: Use regex to split strings, rather than simple string matching



# Finding All

1. "6708 Pine Street, Omaha, NE 68182"
2. "1600 Pennsylvania Ave NW, Washington, DC 20006"
3. "261 S 800 E\nSalt Lake City, UT 84102"

```
myexp = r'(?:(?:\n|, )(.*)?:, )([A-Z]{2}))'  
re.findall(myexp, mystring)
```

Using those `?:` markers, we can omit all of the groups (and ugly separating characters) that we are not interested in collecting.

# For Lab/Homework

See Assignment 12 posted to canvas for details. We will be parsing work emails collected during the Enron investigation.

If you want some great reference for a cheatsheet on regular expression, check out

<http://pycon2017.regex.training/index.html>

It also includes a 3-hour training video on using regular expression.