

# ITEC 1150 Week 10

## Chapter 7

PATTERN MATCHING WITH REGULAR EXPRESSIONS

# What Are Regular Expressions?

- ▶ A **Regular Expression** (often abbreviated **regex**) is a way of expressing patterns for matching or searching in strings.
- ▶ A simple (and only partially correct) regex for checking if an email address is valid might look like this:

`[ \w\.\ ]+@ \w+ \. \w+`

Don't worry, this weird string will be explained later in the lecture.

- ▶ Regular expressions are somewhat like their own programming language, but once you learn regex syntax, you can use regexes in almost any programming language.
- ▶ You can even use them in the find/replace feature of many applications (such as PyCharm!)

# Why Use Regular Expressions?

Consider this source text:

**"You can call Batman at 123-456-7890 or at (098)765-4321"**

How can your program recognize and extract both phone numbers from the source text?

One way would be to write a long program to do that, which would be complicated and hard to write and maintain (see the beginning of [Chapter 7](#) in the book).

Another, better way would be to create and use a regex.

# Using Regular Expressions in Python

- ▶ There are two different things to learn about using regular expressions in Python.
  - ▶ How to write a regular expression that matches what you want to check
  - ▶ How to tell Python what to do with your regular expression

# Writing Regular Expressions

HOW TO WRITE A REGULAR  
EXPRESSION THAT MATCHES  
WHAT YOU WANT TO CHECK

# Resources For Regular Expressions



In addition to the book, these sites are good references for Regular Expressions:

- ▶ <https://regex101.com/>
- ▶ <https://docs.python.org/3/howto/regex.html>
- ▶ [https://www.tutorialspoint.com/python3/python\\_reg\\_expressions.htm](https://www.tutorialspoint.com/python3/python_reg_expressions.htm)
- ▶ <https://www.regular-expressions.info/>

# Step 1: Analyzing the Pattern

- ▶ The first thing we need to do to create a regular expression is to figure out what our pattern of characters is. Let's use an email address as an example:

first.last@minnstate.edu

- ▶ We know that an email address starts with letters before the @ symbol, and sometimes has other characters like periods or underscores.
- ▶ We know there is always an @ symbol between the name part and the domain.
- ▶ We know that after the @ symbol there is always a domain, which has two parts separated by a period. The first part can contain letters, numbers, dashes, and underscores; the second part only contains letters.

# Use The Reference Guide!

It's available in D2L under

Content →

Materials →

Week 10 – Regular Expressions →

RegEx Reference Guide.pdf

Open it now and follow along!



# Character Classes

- ▶ When we need to match certain *types* of characters (for example, any lowercase letter or any digit), we can define **character classes**.
- ▶ A character class is a way of describing what values you want to match. A single character can be used by itself, but if you want to match on a variety of different characters, you can define a class by putting the characters inside square brackets:
  - ▶ `[0123456789]` would match any digit
  - ▶ `[abcdefghijklmnopqrstuvwxyz]` would match any lowercase letter.
  - ▶ `[abc123]` would match any of the characters a, b, c, 1, 2, or 3
- ▶ Obviously, some of those are not very efficient to write, so there are shortcuts.

# Character Classes - Ranges

- ▶ Defining a range of characters can be done by putting the starting and ending values inside square brackets, separated by a dash:
  - ▶ `[0-9]` is the same as `[0123456789]`
  - ▶ `[a-z]` is the same as `[abcdefghijklmnopqrstuvwxyz]`
  - ▶ `[A-Z]` is the same as `[ABCDEFGHIJKLMNOPQRSTUVWXYZ]`
- ▶ These can be combined:
  - ▶ `[A-Za-z0-9]` would match any letter or any digit.
  - ▶ `[A-Za-z]` is the same as `[ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz]`
- ▶ You can also add single characters:
  - ▶ `[A-Za-z0-9_#]` would match any letter, any digit, the underscore, and the pound sign.
- ▶ Warning—some sources will tell you `[A-z]` will match any letter. It will, but it will also match these characters: `[\]^_``

# Character Classes – Shorthand Identifiers

- ▶ For the most common ranges, there are built-in identifiers that can be used:

<code>\d</code>	Any number
<code>\D</code>	Anything but a number
<code>\s</code>	Whitespace (spaces, tabs, newline)
<code>\S</code>	Anything but a space
<code>\w</code>	Any letter, number, or underscore
<code>\W</code>	Anything but a letter, number, or underscore
<code>.</code>	Any character, except for a new line
<code>\b</code>	Space around whole words

# Modifiers

- Now we now how to identify characters, but to turn those into *patterns* we need to be able to identify how the characters repeat:

+	1 or more repetitions
?	0 or 1 repetitions
*	0 or more repetitions
\$	The end of the string
^	The start of the string
	Either/or. For example, <code>x y</code> will match either <code>x</code> or <code>y</code>
[]	A range
{x}	Exactly <code>x</code> repetitions
{x,y}	Between <code>x</code> and <code>y</code> repetitions

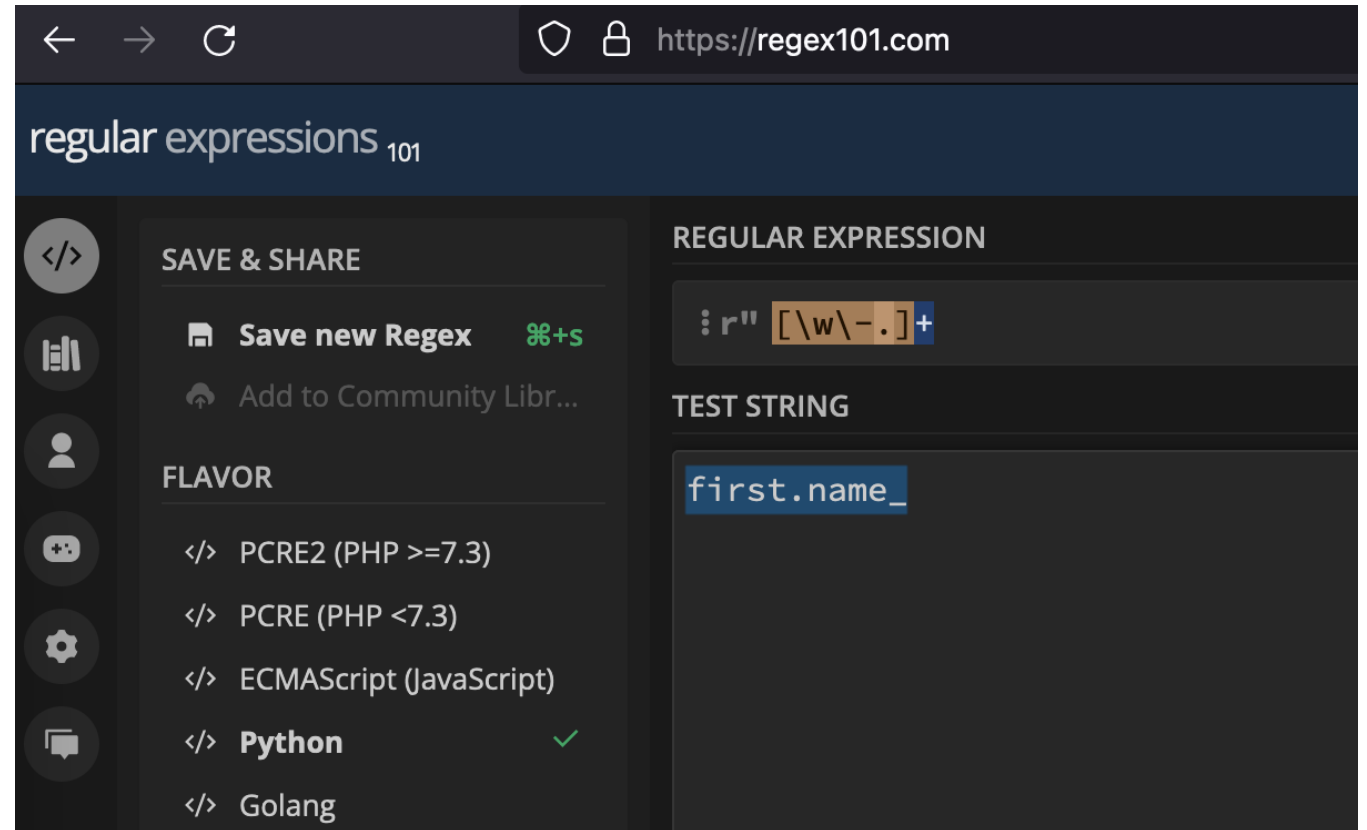
## Step 2: Building Our Regular Expression

first.last@minnstate.edu

- ▶ We know that an email address starts with letters before the @ symbol, and sometimes has other characters like periods or underscores.
- ▶ We can define the first half of the regular expression with the shorthand identifier `\w`, which matches any letter, digit, or underscore
- ▶ We also need to include dashes and periods, so we need to define a character class: `[\w\-.]`
  - ▶ Because the dash is used to define ranges in a character class, we need to escape it with a slash
- ▶ Now we need to specify that our character class must exist one or more times by adding a plus: `[\w\-.]+`

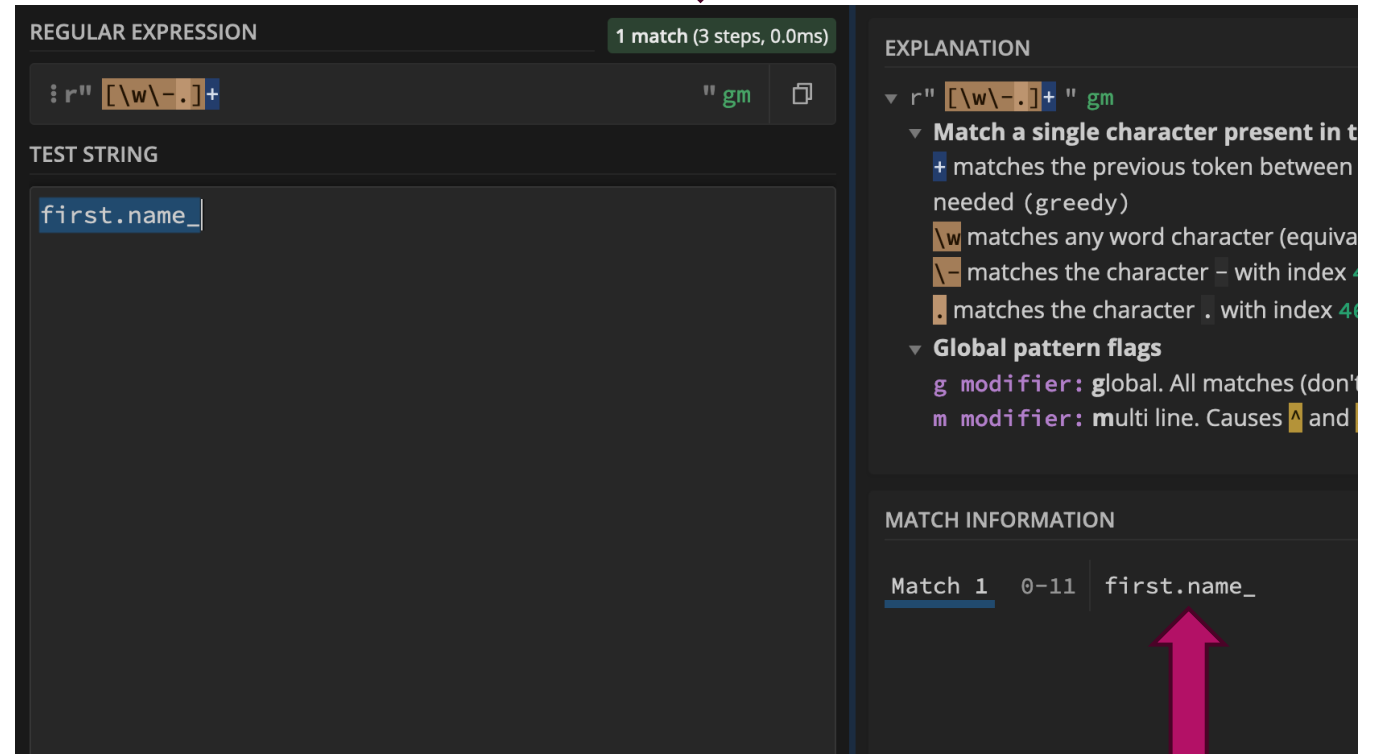
# Testing Our Regex So Far

- ▶ Open <https://regex101.com/> and select Python as the 'flavor' on the left.
- ▶ Enter your regex in the box at the top
- ▶ Enter a string in the area below



# Testing Our Regex So Far (cont.)

- ▶ It tells us there is one match, and what the match is.



REGULAR EXPRESSION 1 match (3 steps, 0.0ms)

`r" [\w\-.]+ "` `gm`

TEST STRING

`first.name_`

EXPLANATION

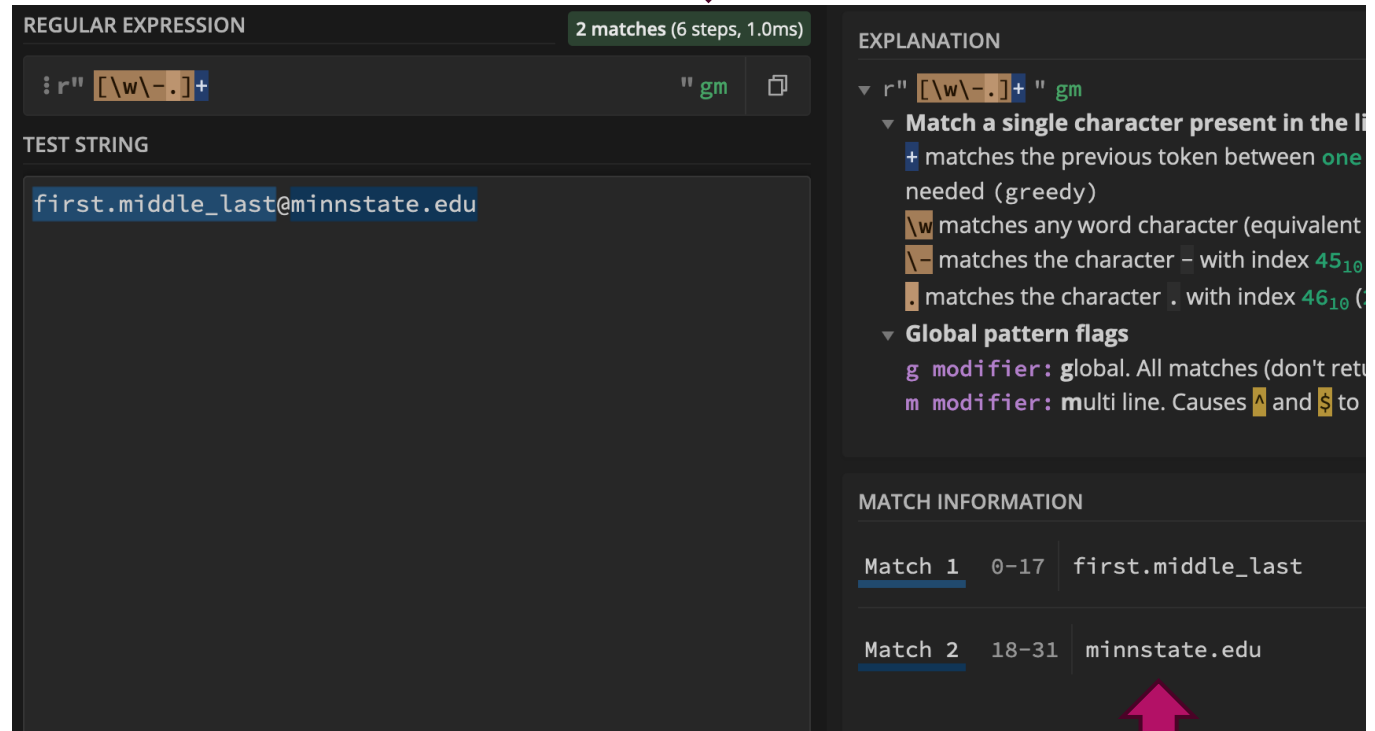
- ▼ `r" [\w\-.]+ "` `gm`
  - ▼ **Match a single character present in the set**
    - `+` matches the previous token between 1 and unlimited times (greedy)
    - `\w` matches any word character (equivalent to `[a-zA-Z0-9_]`)
    - `\-` matches the character `-` with index 4
    - `\.` matches the character `.` with index 4
  - ▼ **Global pattern flags**
    - `g` modifier: global. All matches (don't return just the first)
    - `m` modifier: multi line. Causes `^` and `$` to match at the start and end of each line

MATCH INFORMATION

<u>Match 1</u>	0-11	<code>first.name_</code>
----------------	------	--------------------------

# Testing Our Regex So Far (cont.)

- ▶ If we put in a full email address, note that there are now two matches.
- ▶ Can you explain why?



REGULAR EXPRESSION 2 matches (6 steps, 1.0ms)

`r" [\w\-.]+ "` `gm`

TEST STRING

`first.middle_last@minnstate.edu`

EXPLANATION

▼ `r" [\w\-.]+ "` `gm`

- ▼ **Match a single character present in the li**
  - `+` matches the previous token between **one** needed (greedy)
  - `\w` matches any word character (equivalent
  - `\-` matches the character `-` with index **45**<sub>10</sub>
  - `\.` matches the character `.` with index **46**<sub>10</sub> (
- ▼ **Global pattern flags**
  - `g` **modifier:** global. All matches (don't retu
  - `m` **modifier:** multi line. Causes `^` and `$` to

MATCH INFORMATION

<u>Match 1</u>	0-17	first.middle_last
<u>Match 2</u>	18-31	minnstate.edu



## Step 2: Building Our Regular Expression (cont.)

first.last@minnstate.edu

- ▶ Now that we have the name part of the email working, the next part is the @ symbol. We can just add that to our regex, so now we have

`[\w\-.]+\@`

- ▶ Try it out online. Note that there's one match.

The screenshot shows a web-based regular expression testing interface. The 'REGULAR EXPRESSION' field contains `r" [\w\-.]+\@"` with flags `gm`. The 'TEST STRING' field contains `first.middle_last@minnstate.edu`. A status bar indicates '1 match (3 steps, 1.0ms)'. The 'EXPLANATION' panel on the right details the match: `+` matches the previous token between `on` needed (greedy), `\w` matches any word character (equivalen), `-` matches the character `-` with index `451`, `.` matches the character `.` with index `4610`, and `@` matches the character `@` with index `6410` (4). It also lists 'Global pattern flags' with `g` modifier: global. All matches (don't re) and `m` modifier: multi line. Causes `^` and `$` to

The 'MATCH INFORMATION' table at the bottom shows:

Match	Index	Match
Match 1	0-18	first.middle_last@

## Step 2: Building Our Regular Expression (cont.)

first.last@minnstate.edu

- ▶ The last part is the domain. We know the first part of the domain can be made up of letters, numbers, underscores, and dashes, so we can use the same regex piece we did for the name: `[\w\-.]+\.`
- ▶ Then we need the period (it needs to be escaped!): `[\w\-.]+\@[\w\-.]+\.`
- ▶ Finally, we need the top-level domain, which can only be letters, so we should use a range: `[\w\-.]+\@[\w\-.]+\.[A-Za-z]+\.`

## Test It Out!

- ▶ Your regex needs to match valid email addresses but should not match invalid addresses!
- ▶ Try out both to make sure!

REGULAR EXPRESSION 1 match (10 steps, 1.0ms)

`["r" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`

TEST STRING

`first.middle_last@minnstate.edu`

EXPLANATION

- ▼ **Match a single character present in the list below** `["w" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`
  - `+` matches the previous token between **one** and **unlimited** needed (greedy)
  - `\w` matches any word character (equivalent to `[a-zA-Z0-9_]`)
  - `\-` matches the character `-` with index `4510` (`2D16` or `558`)
  - `\.` matches the character `.` with index `4610` (`2E16` or `568`)
  - `@` matches the character `@` with index `6410` (`4016` or `1008`)
- ▼ **Match a single character present in the list below** `["w" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`
  - `+` matches the previous token between **one** and **unlimited** needed (greedy)

MATCH INFORMATION

Match	Index	String
1	0-31	first.middle_last@minnstate.edu

REGULAR EXPRESSION no match (90 steps, 1.0ms)

`["r" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`

TEST STRING

`first.middle_last@minnstateedu`

EXPLANATION

- ▼ **Match a single character present in the list below** `["w" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`
  - `+` matches the previous token between **one** and **unlimited** needed (greedy)
  - `\w` matches any word character (equivalent to `[a-zA-Z0-9_]`)
  - `\-` matches the character `-` with index `4510` (`2D16` or `558`)
  - `\.` matches the character `.` with index `4610` (`2E16` or `568`)
  - `@` matches the character `@` with index `6410` (`4016` or `1008`)
- ▼ **Match a single character present in the list below** `["w" "\w\.-]+\@["\w\.-]+\.[A-z]+" gm`
  - `+` matches the previous token between **one** and **unlimited** needed (greedy)

MATCH INFORMATION

Your regular expression does not match the subject string.

# Using Regular Expressions in Python

HOW TO TELL PYTHON WHAT  
TO DO WITH YOUR REGULAR  
EXPRESSION

# Using Regular Expressions in Python

Once you have a regex, the following statements and functions are what you need to use it in Python:

<code>import re</code>	Imports the regex module - put at the top of your program
<code>re.compile</code>	Turns your regex string into a regex object Python can use to search
<code>search()</code>	Returns a matched object from the source string or returns <b>None</b> if no match found.
<code>match()</code>	Returns matched text & more
<code>findall()</code>	Returns a list of all matches
<code>group()</code>	Deals with groupings like area code and #

# Compiling a Regex

- ▶ Create a file called `ch7_exercises.py` to run the lesson code.
- ▶ First, we import the `re` module.
- ▶ Next, we call `re.compile()` with our regex string.

```
1. import re # import the regex module
```

```
2. phone_regex = re.compile(r'\d\d\d.\d\d\d.\d\d\d\d') # creates a regex pattern
```

- ▶ Note the 'r' at outside the regex string? That's important because it prevents Python from processing the escape character "\" inside our regex. We want our string to contain "\"d", not an 'escaped d'.

r raw string  
\d any digit  
.  
any single character

# Searching With A Regex

- ▶ Now that we've compiled our regex, we can call the `search()` method on it.
- ▶ `search()` will only find the *first* match, so the example has to split the string to find both phone numbers:

```
1. import re # import the regex module
2. phone_regex = re.compile(r'\d\d\d\d\d\d\d\d') # creates a regex pattern
3. source1 = phone_regex.search('call Batman at 123-456-7890 ')
4. source2 = phone_regex.search('or at (098)765-4321')
5. print('Phone Number 1: ' + source1.group())
6. print('Phone Number 2: ' + source2.group())
```

# Returns All Matches As List With `findall()`

- ▶ The `findall()` method will return every match in a larger searched string, as a list.

```
1. source = 'You can call Batman at 123-456-7890 or at (098)765-4321'
2. phone_regex = re.compile(r'\d\d\d\d\d\d\d\d') # same as before
3. source_list = phone_regex.findall(source)
4. print ('Here is the list of phone numbers: ', source_list)
```

- ▶ Run this, and you should see the response:

```
Here is the list of phone numbers: [' 123-456-7890', '(098)765-4321']
```

- ▶ Why does the first list item have a leading space? How can we remove it?
- ▶ As usual, when you have a list to work with, you can use a loop to operate on the items.



# Checking If A Match Was Found

- ▶ If your search doesn't match anything and you try to get the `group()`, you'll receive an error.
- ▶ Here's how you can check if a match was found:

```
1. source_string = 'Albert Einstein'
2. print('The text we are searching in is called the "source string":', source_string)
3. search_string = input('Enter the text we are searching for, called the "search string": ')
4. match = re.search('.*'+search_string, source_string) # matches all characters up to and including search term
5. print('Match info: ', match) # match will either be None or a more complex object with results
6. if match: # None evaluates to False, while other values evaluate to True, so we can treat it like a boolean
7.     print('Match text or None :', match.group()) # limit result to match text or None w/the group() method
8. else:
9.     print(search_string, 'not found')
```

- ▶ Run this code & try searching for 'Al', 'bert', 'e', 'i', 'X', and more...

# Common Uses Of Group Method

- This phone regex is a little different than before – let's run this code to see what happens

```
1. phone_regex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
2. match = phone_regex.search('You can call Batman at 123-456-7890')
3. print('The source we are searching in is: You can call Batman at 123-456-7890 ')
4. print(r'Pattern: .(\d\d\d)-(\d\d\d-\d\d\d\d)')
5. print('group 1:', match.group(1))
6. print('group 2:', match.group(2))
7. print('group (not specified):', match.group())
8. print('group 0:', match.group(0))
9. print('groups:', match.groups()) # what data type is returned?
```

Adding parentheses ( ) creates groups in the pattern. In this pattern, the area code and phone number are in separate groups.

# MORE EXAMPLES

# Matching Optional Characters Or Groups With ?

- ▶ Use the ? (question mark) when part of the pattern is optional.
- ▶ Finding a phone number with or without the area code and leading parentheses.
  1. `phone_regex = re.compile(r'.?(\d\d\d)?-?(\d\d\d-\d\d\d\d)')` # pattern with optional characters and group
  2. `print(r'Pattern: .?(\d\d\d)?-?(\d\d\d-\d\d\d\d)')` # printed for reference
  3. `source3= phone_regex.search('You can call Batman at 123-456-7890.')`
  4. `print('Source text: You can call Batman at 123-456-7890')` # printed for reference
  5. `print(source3.group())`
  6. `source4 = phone_regex.search('You can call Batman at 456-7890.')`
  7. `print('Source text: You can call Batman at 456-7890.')` # printed for reference
  8. `print(source4.group().strip())` # notice the difference it makes in output to add .strip()

Let's add code to make the 2<sup>nd</sup> hyphen optional & test it.

# Matching multiple Terms with | character

- Use the | (pipe) symbol like an "or" when you are testing for the existence of several possible expressions.

```
1. hero_regex = re.compile(r'Batman|Robin') # establish the pattern
2. source_hero1 = hero_regex.search('Batman and Robin') # provide a source to search in
3. print(source_hero1.group()) # matches the first term found - Batman
4. source_hero2 = hero_regex.search('Robin and Batman') # provide a different source to search in
5. print(source_hero2.group()) # again, matches the first term found - this time it's Robin
```

- You can still use findall() to find a list of all matches, or in this case, both Batman and Robin

```
1. print(hero_regex.findall('Batman and Robin'))
```

# Matching One Or More Occurrence With +

- ▶ Match one or more occurrences of the preceding group by using the plus character
- ▶ Run this code. First the source has 1 "wo", then 3 "wo"s, then no "wo"s

```
1. bat_regex = re.compile(r'Bat(wo)+man') # establish pattern with one or more wo's
2. source_bat2 = bat_regex.search('Where does Batwoman live') # source to search in
3. print('Source includes wo:', source_bat2.group())
4. source_bat3 = bat_regex.search('Where does Batwowowoman live') # source to search in
5. print('Source has LOTS of wo\'s:', source_bat3.group())
6. source_bat1 = bat_regex.search('Where does Batman live') # source to search in
7. print('Source does NOT have wo:', source_bat1) # notice what's returned!
```

# Matching zero or more occurrences with \*

- ▶ Match zero or more occurrences of the preceding group by using the star character
- ▶ Run this code. First the string has no "wo", then 1 "wo", then 3 "wo"s

```
1. bat_regex = re.compile(r'Bat(wo)*man') # establish pattern with zero or more wo's
2. source_bat1 = bat_regex.search('Where does Batman live') # source to search in
3. print('Source has Batman WITHOUT the wo:', source_bat1.group())
4. source_bat2 = bat_regex.search('Where does Batwoman live') # source to search in
5. print('Source has Batman WITH the wo:', source_bat2.group())
6. source_bat3 = bat_regex.search('Where does Batwowowoman live') # source to search in
7. print('Source has Batman with LOTS of wo\'s:', source_bat3.group())
```

# ... and there is so much more!

- ▶ If you work through the PPT and book Chapter 7 exercises, while thinking of possible related applications in the real world, you will probably realize that regular expressions can be a huge and sometimes very complex subject.
- ▶ Entire jobs may revolve around creation, adaptation, perfection & use of RE!
- ▶ This introduction gets you started and will help you see the difference between writing a lot of code to validate input, versus writing a short function with a regular expression to do the same thing.
- ▶ Good luck!