Computer Programming I

Binnur Kurt, PhD

BAU
Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 11

REGULAR EXPRESSIONS

Introduction to Regular Expressions

> Sometimes you'll need to recognize patterns in text, like phone numbers, e-mail addresses, ZIP Codes, web page addresses, Social Security numbers and more.

> A regular expression string describes a search pattern for matching characters in other strings.

> Regular expressions can help you extract data from unstructured text, such as social media posts.

> They're also important for ensuring that data is in the correct format before you attempt to process it.

Validating Data

> Before working with text data, you'll often use regular expressions to ***validate the data***.

> For example, you can check that:

- A U.S. ZIP Code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775).
- A string last name contains only letters, spaces, apostrophes and hyphens.
- An e-mail address contains only the allowed characters in the allowed order.
- A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers that can be used in each group of digits.

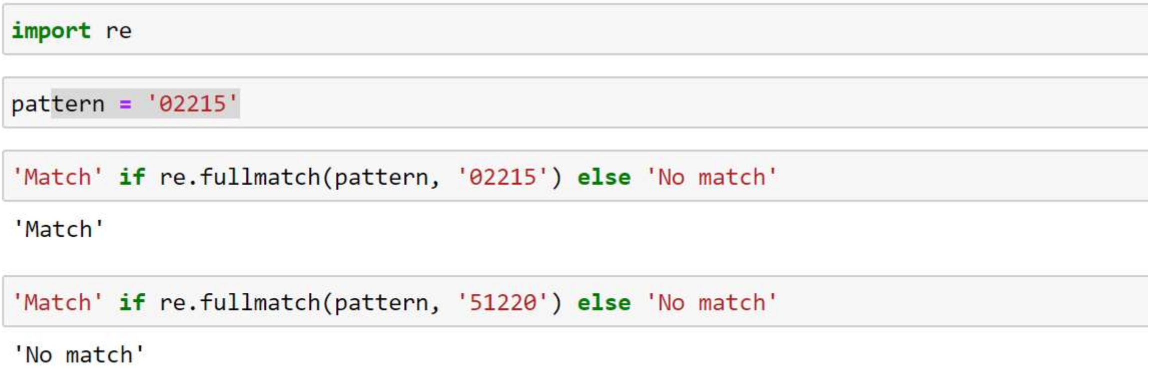
Other Uses of Regular Expressions

> In addition to validating data, regular expressions often are used to:

- Extract data from text (known as scraping)
  - For example, locating all URLs in a web page.
- Clean data
  - For example, removing data that’s not required, removing duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, dealing with outliers and more.
- Transform data into other formats
  - For example, reformatting data that was collected as tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format.

`re` Module and Function `fullmatch`

> To use regular expressions, import the Python Standard Library’s `re` module:



```
import re
```

```
pattern = '02215'
```

```
'Match' if re.fullmatch(pattern, '02215') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch(pattern, '51220') else 'No match'
```

```
'No match'
```

Metacharacters, Character Classes and Quantifiers

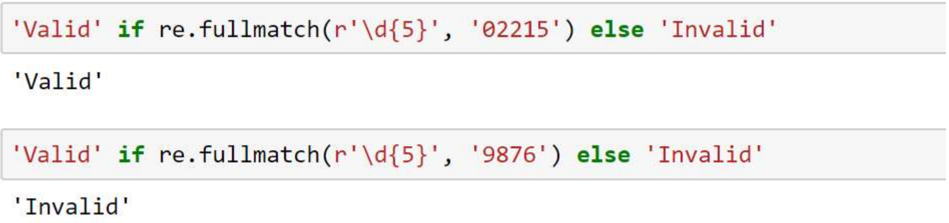
- Regular expressions typically contain various special symbols called **metacharacters**

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

- The **\ metacharacter** begins each of the predefined **character classes**, each matching a specific set of characters

Metacharacters, Character Classes and Quantifiers

- Validate a five-digit ZIP Code



```
'Valid' if re.fullmatch(r'\d{5}', '02215') else 'Invalid'
```

```
'Valid'
```

```
'Valid' if re.fullmatch(r'\d{5}', '9876') else 'Invalid'
```

```
'Invalid'
```

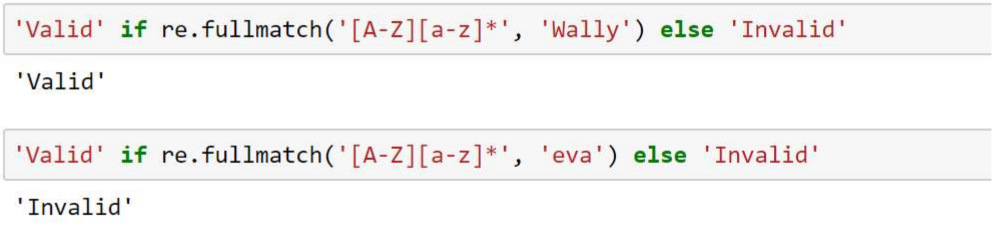
Other Predefined Character Classes

Character class	Matches
`\d`	Any digit (0–9).
`\D`	Any character that is *not* a digit.
`\s`	Any whitespace character (such as spaces, tabs and newlines).
`\S`	Any character that is *not* a whitespace character
`\w`	Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore
`\W`	Any character that is *not* a word character.

Custom Character Classes

- Square brackets, `[]`, define a custom character class that matches a single character.
- Examples
  - [aeiou] matches a lowercase vowel
  - [A-Z] matches an uppercase letter, [a-z] matches a lowercase letter
  - [a-zA-Z] matches any lowercase or uppercase letter

Custom Character Classes: Examples



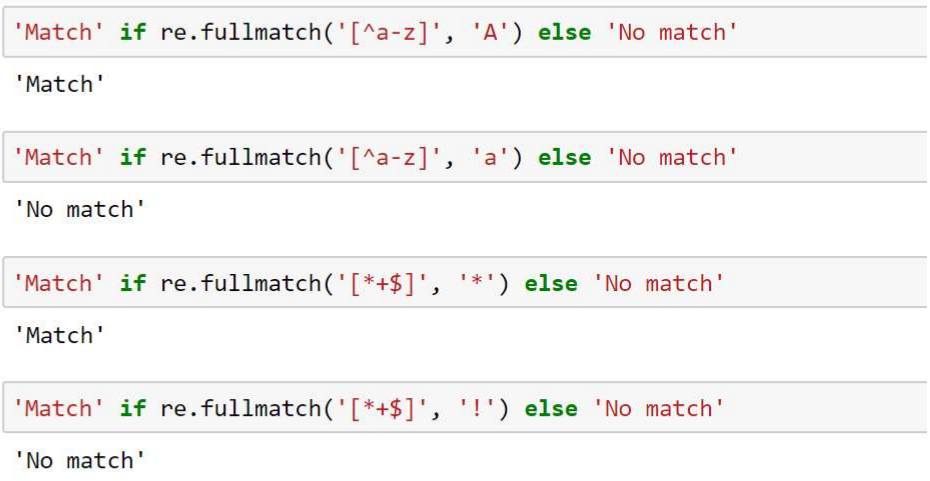
```
'Valid' if re.fullmatch('[A-Z][a-z]*', 'Wally') else 'Invalid'
```

```
'Valid'
```

```
'Valid' if re.fullmatch('[A-Z][a-z]*', 'eva') else 'Invalid'
```

```
'Invalid'
```

Custom Character Classes: Examples



```
'Match' if re.fullmatch('[^a-z]', 'A') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch('[^a-z]', 'a') else 'No match'
```

```
'No match'
```

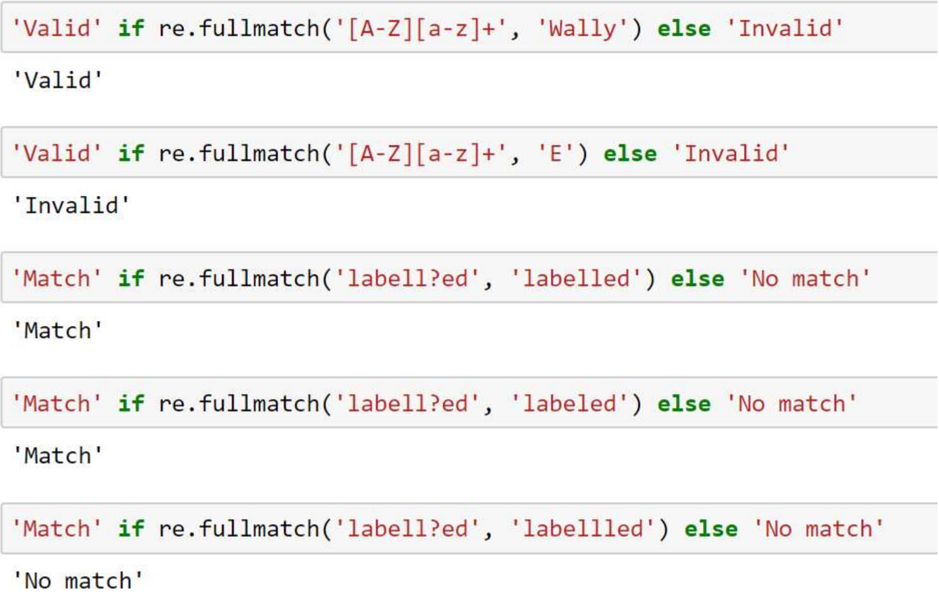
```
'Match' if re.fullmatch('[*+$]', '*') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch('[*+$]', '!') else 'No match'
```

```
'No match'
```

Custom Character Classes: Examples



```
'Valid' if re.fullmatch('[A-Z][a-z]+', 'Wally') else 'Invalid'
```

```
'Valid'
```

```
'Valid' if re.fullmatch('[A-Z][a-z]+', 'E') else 'Invalid'
```

```
'Invalid'
```

```
'Match' if re.fullmatch('labell?ed', 'labelled') else 'No match'
```

```
'Match'
```

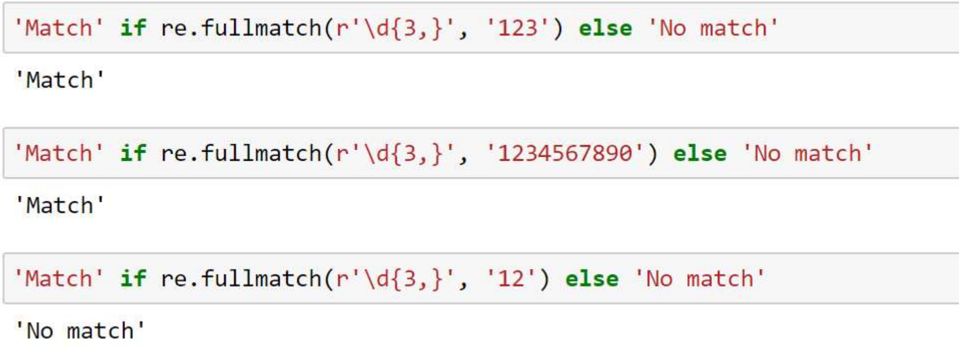
```
'Match' if re.fullmatch('labell?ed', 'labeled') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch('labell?ed', 'labellled') else 'No match'
```

```
'No match'
```

Custom Character Classes: Examples



```
'Match' if re.fullmatch(r'\d{3,}', '123') else 'No match'
```

```
'Match'
```

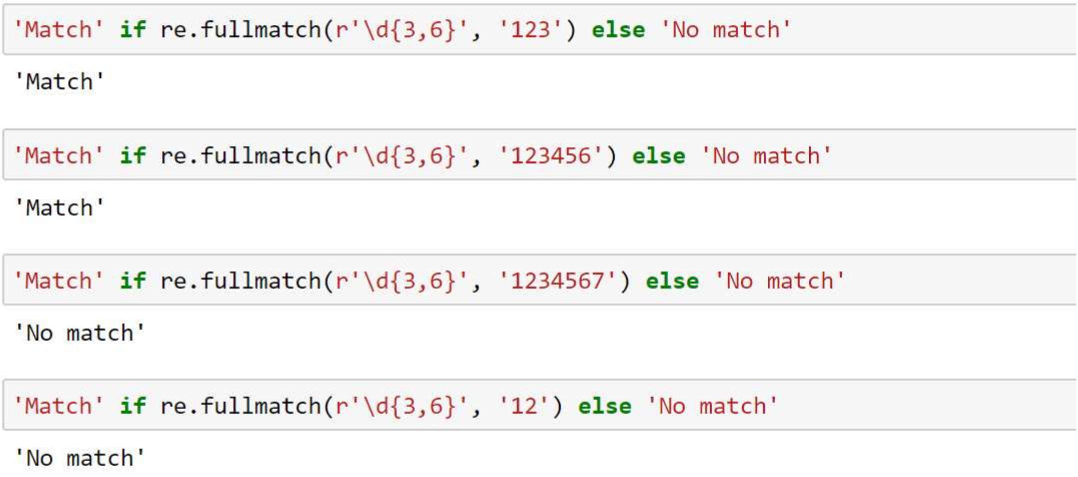
```
'Match' if re.fullmatch(r'\d{3,}', '1234567890') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch(r'\d{3,}', '12') else 'No match'
```

```
'No match'
```

Custom Character Classes: Examples



```
'Match' if re.fullmatch(r'\d{3,6}', '123') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch(r'\d{3,6}', '123456') else 'No match'
```

```
'Match'
```

```
'Match' if re.fullmatch(r'\d{3,6}', '1234567') else 'No match'
```

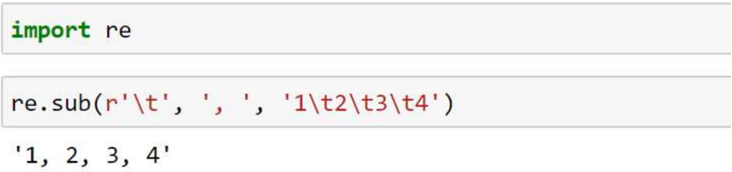
```
'No match'
```

```
'Match' if re.fullmatch(r'\d{3,6}', '12') else 'No match'
```

```
'No match'
```

Function sub—Replacing Patterns

> The **re** module's **sub function** replaces ***all occurrences*** of a pattern with the replacement text you specify



```
import re
```

```
re.sub(r'\t', ', ', '1\t2\t3\t4')
```

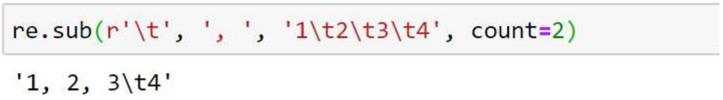
```
'1, 2, 3, 4'
```

> The **sub** function receives three required arguments:

- the pattern to match (the tab character `'\t'`)
- the replacement text (`', '`)
- the string to be searched (`'1\t2\t3\t4'`)

Function sub—Replacing Patterns

> The keyword argument **count** can be used to specify the maximum number of replacements:

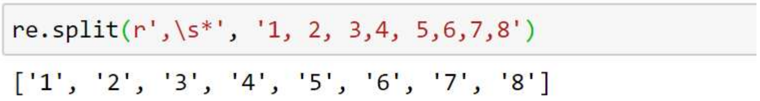


```
re.sub(r'\t', ', ', '1\t2\t3\t4', count=2)
```

```
'1, 2, 3\t4'
```

Function **split**

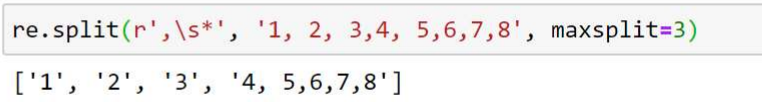
> The **split** function tokenizes a string, using a regular expression to specify the ***delimiter***, and returns a list of strings.



```
re.split(r',\s*', '1, 2, 3,4, 5,6,7,8')
```

```
['1', '2', '3', '4', '5', '6', '7', '8']
```

> Use the keyword argument **maxsplit** to specify the maximum number of splits:

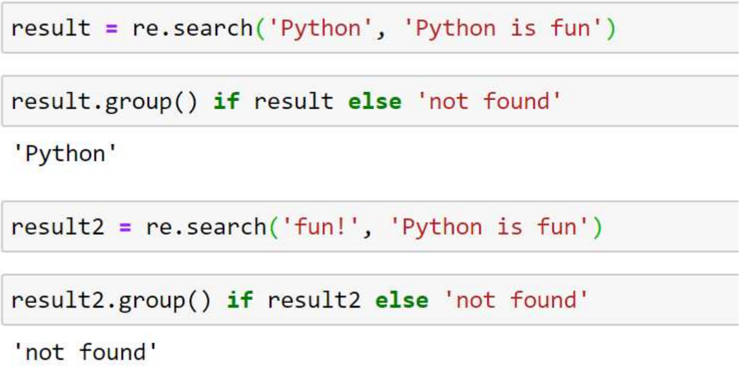


```
re.split(r',\s*', '1, 2, 3,4, 5,6,7,8', maxsplit=3)
```

```
['1', '2', '3', '4, 5,6,7,8']
```

Finding the First Match Anywhere in a String

- Function **search** looks in a string for the first occurrence of a substring that matches a regular expression and returns a **match object** (of type **SRE_Match**) that contains the matching substring.
- The match object's **group** method returns that substring:



```
result = re.search('Python', 'Python is fun')
```

```
result.group() if result else 'not found'
```

```
'Python'
```

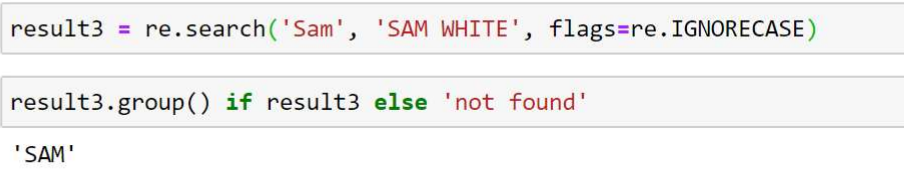
```
result2 = re.search('fun!', 'Python is fun')
```

```
result2.group() if result2 else 'not found'
```

```
'not found'
```

Ignoring Case with the Optional `flags` Keyword Argument

- Many **re** module functions receive an optional **flags** keyword argument that changes how regular expressions are matched.
- For example, matches are case sensitive by default, but by using the **re** module's **IGNORECASE** constant, you can perform a case-insensitive search



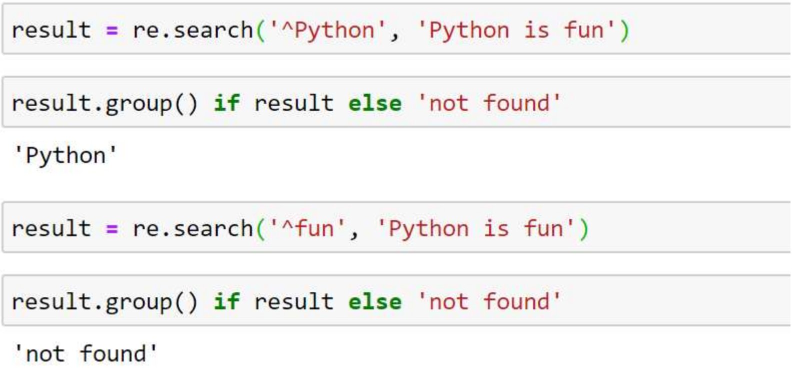
```
result3 = re.search('Sam', 'SAM WHITE', flags=re.IGNORECASE)
```

```
result3.group() if result3 else 'not found'
```

```
'SAM'
```

Restricting Matches to the Beginning or End of a String

> The ^ metacharacter at the beginning of a regular expression is an anchor indicating that the expression matches only the beginning of a string



```
result = re.search('^Python', 'Python is fun')
```

```
result.group() if result else 'not found'
```

```
'Python'
```

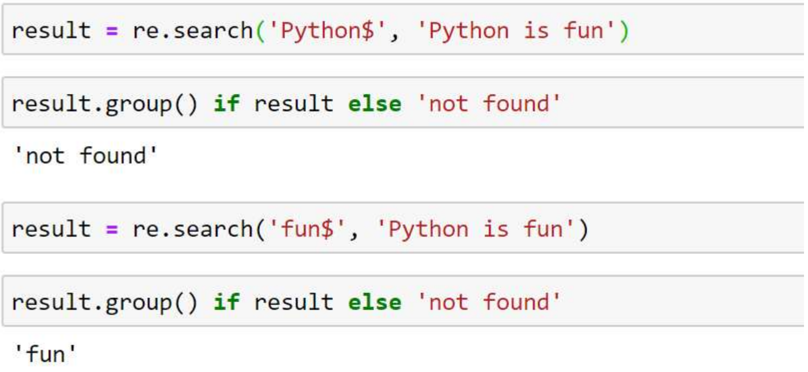
```
result = re.search('^fun', 'Python is fun')
```

```
result.group() if result else 'not found'
```

```
'not found'
```

Restricting Matches to the Beginning or End of a String

> The $ metacharacter at the end of a regular expression is an anchor indicating that the expression matches only the end of a string



```
result = re.search('Python$', 'Python is fun')
```

```
result.group() if result else 'not found'
```

```
'not found'
```

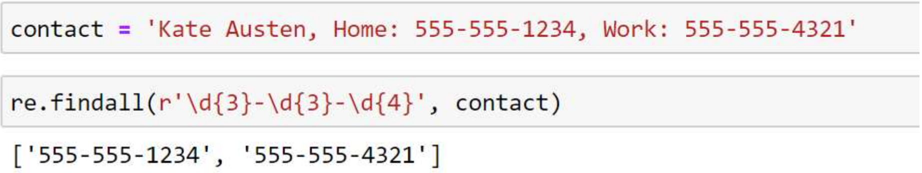
```
result = re.search('fun$', 'Python is fun')
```

```
result.group() if result else 'not found'
```

```
'fun'
```

Finding All Matches in a String

> Function **findall** finds every matching substring in a string and returns a list of the matching substrings.



```
contact = 'Kate Austen, Home: 555-555-1234, Work: 555-555-4321'
```

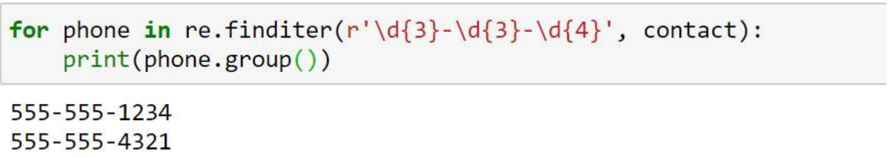
```
re.findall(r'\d{3}-\d{3}-\d{4}', contact)
```

```
['555-555-1234', '555-555-4321']
```

Finding All Matches in a String

> Function **finditer** works like **findall**, but returns a ***lazy iterable*** of match objects.

> For large numbers of matches, using **finditer** can save memory because it returns one match at a time, whereas **findall** returns all the matches at once

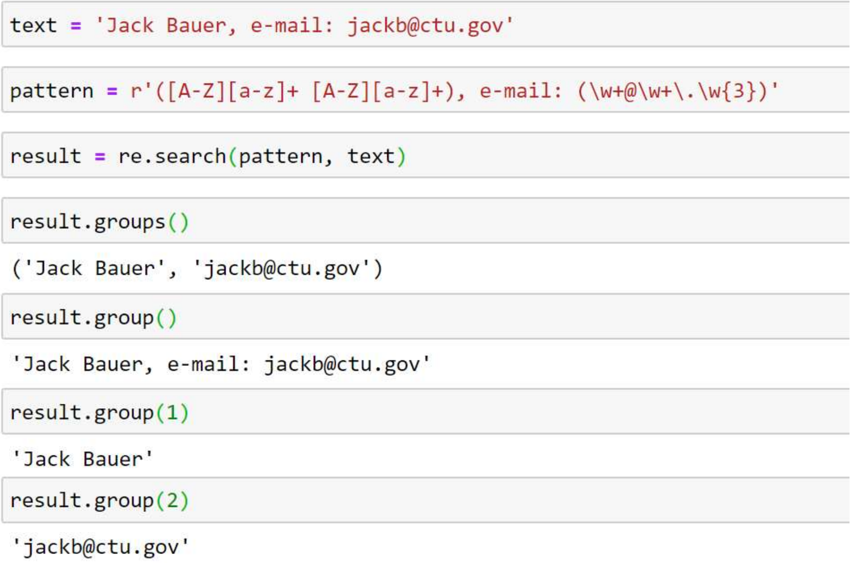


```
for phone in re.finditer(r'\d{3}-\d{3}-\d{4}', contact):
    print(phone.group())
```

```
555-555-1234
555-555-4321
```

Capturing Substrings in a Match

> You can use metacharacters, (and), to capture substrings in a match



```
text = 'Jack Bauer, e-mail: jackb@ctu.gov'
```

```
pattern = r'([A-Z][a-z]+ [A-Z][a-z]+), e-mail: (\w+@\w+\.\w{3})'
```

```
result = re.search(pattern, text)
```

```
result.groups()
```

```
('Jack Bauer', 'jackb@ctu.gov')
```

```
result.group()
```

```
'Jack Bauer, e-mail: jackb@ctu.gov'
```

```
result.group(1)
```

```
'Jack Bauer'
```

```
result.group(2)
```

```
'jackb@ctu.gov'
```