

Strings

Lecture 6

Department of Computer Science and Engineering, ITER
Siksha 'O' Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.



Contents

- 1 Introduction
- 2 Slicing
- 3 Membership
- 4 Built-in Functions on Strings
- 5 String Processing Examples

- A string is a sequence of characters represented using single, double, or triple quotes.

```
>>> message = 'Hello Gita'
```

- Triple quotes are typically used for strings that span multiple lines.
- `len` function finds the length of a string

```
>>> len(message)  
10
```

Introduction (Cont.)

Individual characters within a string are accessed using a technique known as indexing.

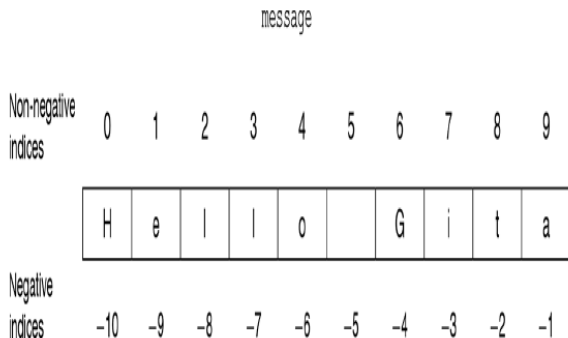


Figure 1: Indexing of variable **message**

Introduction (Cont.)

- An index is specified within the square brackets to access individual characters in a string via indices, for example:

```
>>> message[6]
```

```
'G'
```

```
>>> index = len(message) - 1
```

```
>>> message[index]
```

```
'a'
```

- The negative indices range from - (length of the string) to -1.
- The entire range of valid indices would be $\{-10, -9, \dots, -1, 0, 1, \dots, 9\}$.

```
>>> message[-1]
```

```
'a'
```

```
>>> message[-index]
```

```
'e'
```

Introduction (Cont.)

```
>>> message[15]
```

Traceback (most recent call last):

File "<pyshell#17>", line 1, in <module>

```
message[15]
```

IndexError: string index out of range

Strings in Python are immutable.

```
>>> message[6] = 'S'
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

```
message[6] = 'S'
```

TypeError: 'str' object does not support item assignment

Introduction (Cont.)

```
>>> 'Computer' + ' Science'  
'Computer Science'  
>>> 'Hi'*3  
'HiHiHi'
```

max() and min() are used to find maximum and minimum respectively of values.

```
>>> max('AZ', 'C', 'BD', 'BT')  
'C'  
>>> min('BD', 'AZ', 'C')  
'AZ'  
>>> max('hello', 'How', 'Are', 'You', 'sir')  
'sir'
```

Slicing

- To retrieve a sub-string, also called a slice, from a string.

```
>>> message = 'Hello Sita'
```

```
>>> message[0:5]
```

```
'Hello'
```

```
>>> message[-10:-5]
```

```
'Hello'
```

- Python assumes 0 as the default value of start, and length of the string as the default value of end.

```
>>> message[:5]
```

```
'Hello'
```

```
>>> message[5:]
```

```
' Sita'
```

```
>>> message[:]
```

```
"Hello Sita"
```


Slicing (Cont.)

```
>>> message[:5] + message[5:]  
'Hello Sita'  
>>> message[:15]  
'Hello Sita'  
>>> message[15:]  
''  
>>> message[:15] + message[15:]  
'Hello Sita'  
>>> message[6:None]  
'Sita'
```

Membership

- Python allows us to check for membership of the individual characters or sub-strings in strings using *in* operator.
- The expression `s in str1` yields `True` or `False` depending on whether `s` is a sub-string of `str1`, for example:

```
>>> 'h' in 'hello'
```

```
True
```

```
>>> 'ell' in 'hello'
```

```
True
```

```
>>> 'h' in 'Hello'
```

```
False
```

- For loop can be used to iterate over each element in a sequence.

Built-in Functions on Strings (Cont.)

Functions	Explanation
s.count(str1)	counts number of times string str1 occurs in the string s
s.find(str1)	Returns index of the first occurrence of the string str1 in string s, and returns -1 if str1 is not present in string s
s.rfind(str1)	Returns index of the last occurrence of string str1 in string s, and returns -1 if str1 is not present in string s
s.capitalize(str1)	Returns a string that has first letter of the string s in uppercase and rest of the letters in lower-case

Built-in Functions on Strings (Cont.)

<code>s.title()</code>	returns a string that has first letter of every word in the string <code>s</code> in uppercase and rest of the letters in the lowercase
<code>s.lower()</code>	returns a string that has all uppercase letters in string <code>s</code> converted into corresponding lowercase letters
<code>s.upper()</code>	returns a string that has all lowercase letters in string <code>s</code> converted into corresponding uppercase letters
<code>s.swapcase()</code>	returns a string that has all lowercase letters in string <code>s</code> converted into uppercase letters and vice-versa

Built-in Functions on Strings (Cont.)

<code>s.islower()</code>	returns True if all alphabets in string s (comprising atleast one alphabet) are in lowercase, else returns False
<code>s.isupper()</code>	returns True if all alphabets in string s (comprising atleast one alphabet) are in uppercase, else returns False
<code>s.istitle()</code>	returns True if string s is in the title case, i.e., only first letter of each word is capitalized and the string s contains atleast one alphabet, and returns False otherwise
<code>s.replace(str1, str2)</code>	returns a string that has every occurrence of string str1 in s replaced by string str2

Built-in Functions on Strings (Cont.)

<code>s.strip()</code>	returns a string that has whitespaces in <code>s</code> removed from the beginning and the end
<code>s.lstrip()</code>	returns a string that has whitespaces in <code>s</code> removed from the beginning
<code>s.rstrip()</code>	returns a string that has whitespaces in <code>s</code> removed from the end
<code>s.split(delimiter)</code>	returns a list formed by splitting the string <code>s</code> into substrings. The delimiter is used to mark the split points
<code>s.partition (delimiter)</code>	partitions the string <code>s</code> into three parts based on <u>delimiter</u>

Built-in Functions on Strings (Cont.)

<code>s.join(sequence)</code>	returns a string comprising elements of the sequence separated by delimiter <code>s</code>
<code>s.isspace()</code>	returns true if all characters in string <code>s</code> comprise whitespace characters only, i.e., ' ', '\n', and '\t' else False
<code>s.isalpha()</code>	returns true if all characters in string <code>s</code> comprise alphabets only, and False otherwise
<code>s.isdigit()</code>	returns true if all characters in string <code>s</code> comprisedigits only, and false otherwise
<code>s.isalnum()</code>	returns true if all characters in string <code>s</code> comprise alphabets and digits only, and false otherwise

Built-in Functions on Strings (Cont.)

<code>s.startswith(str1)</code>	returns true if string <code>s</code> starts with string <code>str1</code> , and false otherwise
<code>s.endswith(str1)</code>	returns true if string <code>s</code> ends with string <code>str1</code> , and false otherwise
<code>s.encode(encoding)</code>	returns <code>s</code> in an encoded form, based on the given encoding scheme
<code>s.decode(encoding)</code>	returns the decoded string <code>s</code> , based on the encoding scheme

Table 1: String Functions

Built-in Functions on Strings (Cont.)

```
>>> 'Encyclopedia'.count('c')
```

```
2
```

```
>>> colors = 'green, red, blue, red, red, green'  
colors.find('red')
```

```
7
```

```
>>> colors.rfind('red')
```

```
23
```

```
>>> colors.find('orange')
```

```
-1
```

```
>>> 'python IS a Language'.capitalize()
```

```
'Python is a language'
```

```
>>> 'python IS a PROGRAMMING Language'.title()
```

```
'Python Is A Programming Language'
```

Built-in Functions on Strings (Cont.)

```
>>> email1 = 'geek@gmail.com'
>>> email2 = 'Geek@gmail.com'
>>> email1 == email2
False
>>> email1.lower() == email2.lower()
True
>>> email1.upper() == email2.upper()
True
>>> 'pYTHON IS PROGRAMMING LANGUAGE'.swapcase()
'Python is programming language'
>>> 'python'.islower()
True
>>> 'Python'.isupper()
False
```

Built-in Functions on Strings (Cont.)

```
>>> 'Basic Python Programming'.istitle()
True
>>> 'Basic PYTHON Programming'.istitle()
False
>>> '123'.istitle()
False
>>> 'Book 123'.istitle()
True
>>> message = 'Amey my friend, Amey my guide'
>>> message.replace('Amey', 'Vihan')
'Vihan my friend, Vihan my guide'
```

Built-in Functions on Strings (Cont.)

```
>>> ' Hello How are you! '.lstrip()
'Hello How are you! '
>>> ' Hello How are you! '.rstrip()
' Hello How are you!'
>>> ' Hello How are you! '.strip()
'Hello How are you!'
>>> colors = 'Red, Green, Blue, Orange, Yellow, Cyan'
>>> colors.split(',')
['Red', ' Green', ' Blue', ' Orange', ' Yellow', ' Cyan']
>>> colors.split()
['Red,', 'Green,', 'Blue,', 'Orange,', 'Yellow,', 'Cyan']
>>> 'Hello. How are you?'.partition('.')
('Hello', '.', ' How are you?')
```

Built-in Functions on Strings (Cont.)

```
>>> ' '.join(['I', 'am', 'ok'])  
'I > am > ok'  
>>> ' '.join(('I', 'am', 'ok'))  
'I am ok'  
>>> ' '.join("'I', 'am', 'ok'")  
"'>I'>, >>>'>a>m'>,>>>'>o>k'>"  
>>> name = input('Enter your name : ')  
>>> Enter your name : Nikhil  
>>> name.isalpha()  
True  
>>> name = input('Enter your name : ')  
>>> Enter your name : Nikhil Kumar  
>>> name.isalpha()  
False
```

Built-in Functions on Strings (Cont.)

```
>>> mobileN = input('Enter mobile no : ')
Enter mobile no : 1234567890
>>> mobileN.isdigit() and len(mobileN) == 10
True
>>> ' '.isspace()
True
>>> password = input('Enter password : ')
Enter password : Kailash107Ganga
>>> password.isalnum()
True
>>> password = input('Enter password : ')
Enter password : Kailash 107 Ganga
>>> password.isalnum()
False
>>> name = 'Ankita Narain Talwar'
>>> name.endswith('Talwar')
True
```


Counting the Number of Matching Characters in a Pair of Strings

```
def nMatchedChar(str1, str2):
```

```
    """
```

Objective: to count number of occurrences of characters in str1 that are also in str2

Input parameters: str1, str2-string

Return value: count-numeric

```
    """
```

```
    temp1 = str1.lower()
```

```
    temp2 = str2.lower()
```

```
    count=0
```

```
    for ch1 in temp1:
```

```
        #search for ch1 in temp2
```

```
            for ch2 in temp2:
```

```
                if ch1==ch2:
```

```
                    count+=1
```

```
    return count
```


Counting the Number of Matching Characters in a Pair of Strings (Cont.)

```
>>> name1 = 'Ram Rahim'  
>>> name2 = 'SAMARTH RAHI'  
>>> nMatchedChar(name1, name2)  
16
```

Counting the Number of Common Characters in a Pair of Strings

'''

Objective: to count number of occurrences of characters in two strings

Input parameters: str1,str2-string

Return value: count-numeric

'''

Counting the Number of Common Characters in a Pair of Strings

```
def nCommonChar(str1, str2):  
    temp1 = str1.lower()  
    temp2 = str2.lower()  
    count=0  
    for i in range(len(temp1)):  
        ch1=temp1[i]  
        if not(ch1 in temp1[:i]):  
            #if the character has not been encountered earlier  
            for ch2 in temp2:  
                if ch1==ch2:  
                    count+=1  
                    break  
    return count
```

Reversing a String

```
def reverse(str1):
```

```
    """
```

```
        Objective: to reverse a string
```

```
        Input parameters: str1-string
```

```
        Return value: reverseStr-reverse of str1 - string
```

```
    """
```

```
    reverseStr = ""
```

```
    for i in range(len(str1)):
```

```
        reverseStr=str1[i] + reverseStr
```

```
    return reverseStr
```

Reversing a String (Cont.)

using recursion

```
def reverse(str1):
```

```
    """
```

```
        Objective: to reverse a string
```

```
        Input parameters: str1-string
```

```
        Return value: reverse of str1- string
```

```
    """
```

```
    if str1 == ""
```

```
        return str1
```

```
    else:
```

```
        return reverse(str1[1:]) + str1[0]
```

- **Alphabet (Σ):** An alphabet is a non-empty set of symbols
- **String:** A string is a finite sequence of symbols chosen from the alphabet. An empty string "containing no symbols is called null string and is often denoted by λ or ϵ . The length of a string is defined as the number of symbols in the string. The length of the null string is defined to be zero.
- **Language:** It is the set of strings (words) defined over the alphabet that conforms to some predefined pattern, or rule(s).
- A regular language is described by a regular expression
- regular expressions: used to define regular languages

Pattern Matching (Cont.)

- Each symbol of the alphabet defines a regular language, comprising the symbol itself.
- If r is regular expression, $L(r)$ denotes the language described by r .
- λ or ϵ is a regular expression that denotes the language comprising the null string only.
- The language containing no word, not even λ is called null or empty language $\{\}$, and is denoted by the regular expression ϕ .
- If r and s are regular expressions, $r|s$ is also a regular expression and denotes the language $L(r|s) = L(r) \cup L(s)$.
- If r is a regular expression, so is (r) , denoting the same language, i.e. $L(r) = L((r))$. Parentheses are used to enforce precedence of operators in the regular expressions.
- The regular expression rs denotes the language $L(rs) = L(r) L(s)$
- Concatenation has higher precedence than the union operator

Pattern Matching (Cont.)

- Concatenation is not commutative
- The regular expression $(0|1)1$ denotes the language $L((0|1)1) = L((0|1))L(1) = \{0,1\} \{1\} = \{01,11\}$.
- If r is a regular expression, r^* is also a regular expression. $*$ denotes **zero or more** occurrences of the preceding pattern r .
- 0^* defined over alphabet $\Sigma = \{0, 1\}$ defines the language, $L = \{\lambda, 0, 00, 000, 0000, \dots\}$
- Given a pattern r ,

$$r^+$$

denotes the language comprising **one or more** occurrences of strings that match the pattern r .

- For dealing with a regular expression in Python, we need to import the module `re`, which contains functions for handling the regular expressions

Pattern Matching (Cont.)

Symbols used in regular expression	Meaning
*	zero or more occurrences of the preceding pattern
.	exactly one arbitrary character excluding new-line
?	zero or one occurrence of the preceding pattern
+	one or more occurrences of the preceding pattern
{m}	<u>exactly m occurrences of the preceding pattern</u>

Pattern Matching (Cont.)

<code>{m,n}</code>	at least m and at most n occurrences of preceding term. In absence of n, there is no upper bound and in the absence of m, the lower bound is assumed to be zero
<code>{list-of-char}</code>	a single character from list of characters enclosed between []
<code>[.]</code>	matches dot (not an arbitrary character)
<code>[a-z]</code>	a single character in the range a to z
<code>[A-Z]</code>	a single character in the range A to Z

Pattern Matching (Cont.)

[0-9]	A single digit in the range 0 to 9
[^...]	when ^ occurs at the beginning of a list symbols enclosed between [], it denotes a single character not in the list
^	matches beginning of the string
\$	matches end of the string or just before the newline character at the end of the string
r1 r2	<u>regular expression r1 or r2</u>

Pattern Matching (Cont.)

()	Groups pattern elements
\d	any digit
\D	any non-digit character
\s	whitespace character
\S	Non-whitespace character
\w	any alphanumeric character including _
\W	any non-alphanumeric character excluding _

Table 2: symbols used in regular expressions

Pattern Matching (Cont.)

regular expression	set of matched patterns
python	python
{p P}ython	{python, Python}
a*	{ λ , a, aa, aaa, ...}
a+	{, a, aa, aaa,...}
a?	{ λ , a}
[aeiou]	{a,e,i,o,u}
[ab]?	{ λ , a, b}
[ab]*	{ λ , a, b, aa, ab, bb, aa, aab, ...}

Pattern Matching (Cont.)

regular expression	set of matched patterns
<code>\d</code>	<code>{0,1,2,3,4,5,6,7,8,9}</code>
<code>\d{2}</code>	<code>{00,01,02,03,...,99}</code>
<code>\d{2,3}</code>	<code>{00,01,02,...99,000,001,002,...,999}</code>
<code>\D</code>	<code>{a,b,...,z,A,B,...,Z,*, \$, !,}</code>
<code>\w</code>	<code>{a,b,...,z,A,B,...,Z,0,1,2,...,9}</code>
<code>\s</code>	<code>{space, tab, newline, carriage return}</code>
<code>[^a-b]</code>	the set comprising characters other than a and b
<code>(a b)(c)\$</code>	<code>{ac,bc}</code> , there should be no character after c in the string that matches the pattern

Pattern Matching (Cont.)

$^{\wedge}(a)(0 1)^{*}$	$\{a, a0, a1, a00, a01, \dots\}$, a should be first in the string which pattern matches
$a\backslash^{*}b$	$\{a^{*}b\}$, when the backslash character precedes a character with a special meaning, the special meaning of the character is ignored. Example, although the regular expression $a^{*}b$ defines a pattern comprising zero or more occurrences of a, followed by b, the pattern $a\backslash^{*}b$ defines the string 'a*b'

Table 3: Examples of regular expressions and the corresponding languages

Pattern Matching (Cont.)

- The function **search** of this module is used for matching a regular expression in the given string.
- It looks for the first location of a match in the given string.
- If the search is successful, it returns the **matchObject** instance matching the regular expression pattern, otherwise it returns **None**.
- The function **group** of matchObject instance returns the substring that matches the regular expression.

Pattern Matching (Cont.)

Regular Expression	Example
Python	<pre>>>> string1 = 'Welcome to python shell' >>> match = re.search('python', string1) >>> match.group() 'python'</pre>
{p P}ython	<pre>>>> string1 = 'Welcome to Python shell' >>> match = re.search('{p P}ython', string1) >>> match.group() 'Python'</pre>
Shel*	<pre>>>> string1 = 'Python shell' >>> match = re.search('Shel*', string1) >>> match.group() 'Shell'</pre>

Pattern Matching (Cont.)

Shel?

```
>>> string1 = 'Python shell'
>>> match = re.search('Shel?', string1)
>>> match.group()
'Shel'
```

Shel{1,2}

```
>>> string1 = 'Python shelllll'
>>> match = re.search('Shel{1,2}', string1)
>>> match.group()
'Shell'
```

.....

```
>>> string1 = 'Python shell'
>>> if re.search('.....', string1):
    print('String length is greater than or
equal to 5')
String length is greater than or equal to 5
```

Pattern Matching (Cont.)

`^Python`

```
>>> string1 = 'Python is a powerful language'
>>> if(re.search('^Python', string1))
    print('String starts with python')
'String starts with python'
```

`^power`

```
>>> string1 = 'Python is a powerful language'
>>> if(re.search('^power', string1))
    print('String starts with power')
else:
    print('String does not start with power')
```

String does not start with power

`powerful$`

```
>>> string1 = 'Python is a powerful language'
>>> if(re.search('powerful$', string1))
    print('String ends with powerful')
else:
    print('String does not ends with powerful')
String does not ends with powerful
```

Pattern Matching (Cont.)

language\$

```
>>> string1 = 'Python is a powerful language'
>>> if(re.search('language$', string1))
    print('String ends with language')
else:
    print('String does not ends with lan-
    guage')
```

String ends with language

\d\d\d\d\d

```
>>> string1 = 'Roll number is 23456'
>>> match=re.search('\d\d\d\d\d', string1)
>>> match.group()
'23456'
```

\d{5}

```
>>> string1 = 'Roll number is 23456'
>>> match=re.search('\d{5}', string1)
>>> match.group()
'23456'
```

Pattern Matching (Cont.)

`-[0-9]+\.[0-9]+`

```
>>> string1 = 'Decrease in price is -45.89'  
>>> match = re.search('-[0-9]+\.[0-9]+', string1)  
match.group()  
-45.89
```

`\w*`

```
>>> string1 = 'Python Shell'  
>>> match = re.search('\w*', string1)  
match.group()  
'Python'
```

`\w*\s\w*`

```
>>> string1 = 'We used Python Shell'  
>>> match = re.search('\w*\s\w*', string1)  
match.group()  
'We used'
```

`.*`

```
>>> string1 = 'I use **Python**, do you?'  
>>> match = re.search('.*', string1)  
match.group()  
'I use **Python**, do you?'
```

Pattern Matching (Cont.)

`(a(b|c))*`

```
>>> string1 = 'abac12ccaab'
>>> match = re.search('(a(b|c))*', string1)
match.group()
'abac'
```

`(a(b|c))*\d{1,2}c*`

```
>>> string1 = 'abac12ccaab'
>>> match = re.search('(a(b|c))*\d{1,2}c*',
string1))
match.group()
'abac12cc'
```

`\w*\d\d.*b$`

```
>>> string1 = 'abac12ccaab'
>>> match = re.search('\w*\d\d.*b$\s*\w*',
string1))
match.group()
'abac12ccaab'
```

Pattern Matching (Cont.)

<code>(a(b c))*</code>	<pre>>>> string1 = 'abac12ccaab' >>> match = re.search('(a(b c))*', string1) match.group() 'abac'</pre>
<code>(a(b c))+</code>	<pre>>>> string1 = 'abac12ccaab' >>> match = re.search('(a(b c))+', string1) match.group() 'abac'</pre>

Table 4: Python examples of regular expressions

Pattern Matching (Cont.)

To find email ids (pranav.gupta@cs.iitd.ac.in) from a string:

a sequence of alphanumeric characters

denoted by `[a-z0-9]+`

a repeating(0 or more times) sequence of dots followed by alphanumeric characters

denoted by `(\.[a-z0-9]+)*`

@

denoted by `@`

sequence of alphabetic characters

denoted by `[a-z]+`

repeating (1 or more times) sequence of dot followed by alphabetic characters

denoted by `(\.[a-z]+)+`

An email id may be represented using the regular expression

`[a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+.`

Pattern Matching (Cont.)

- Generally, a regular expression is preceded with `r` to denote a raw string.
- Use of a raw string as a regular expression avoids any confusion with some characters that have special meaning in regular expressions.

```
>>> match = re.search(r'[a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com')
```

```
>>> match.group()
```

```
'ram@gmail'
```

Pattern Matching (Cont.)

- **finditer**: retrieving all substrings matching a regular expression.

```
>>> for i in re.finditer(r'[a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com'):  
    print(i.group())
```

```
'ram@gmail.com'
```

```
'pranav.gupta@cs.iitd.ac.in'
```

```
'nik@yahoo.com'
```

```
'raman@gmail.com'
```

Pattern Matching (Cont.)

- searching for words ending with **ing** in a string

```
>>> for i in re.finditer(r'[A-Za-z]+ing', 'Walking down the road, he was  
thinking about the coming years.):
```

```
>>> print(i.group())
```

Walking

thinking

coming

- **findall**: retrieve a list of all the substrings matching a regular expression

```
>>> for i in re.findall(r'[A-Za-z]+ing', 'Walking down the road, he was  
thinking about the coming years.):
```

```
['Walking', 'thinking', 'coming']
```

Pattern Matching (Cont.)

```
>>> message = 'Python:Python is an interactive language. It is  
developed by Guido Van Rossum'  
>>> words = re.findall('\w+', message)  
>>> len(words)  
13  
>>> re.findall(r'([a-z0-9]+(\.[a-z0-9]+)*@[a-z]+(\.[a-z]+)+)',  
'ram@gmail.com, pranav.gupta@cs.iitd.ac.in, nik@yahoo.com,  
raman@gmail.com')  
[('ram@gmail.com', '', '.com'), ('pranav.gupta@cs.iitd.ac.in', '.gupta',  
'in'), ('nik@yahoo.com', '', '.com'), ('raman@gmail.com', '', '.com')]
```

Pattern Matching (Cont.)

- regular expression to extract all the single line comments

#.*

Example:

```
>>> pythonCode = """  
"""
```

Python code to add two numbers.

```
"""  
  
a = 5 #number1  
b = 5 #number2  
Compute addition of two numbers  
c = a + b  
"""  
  
>>> for i in re.finditer(r'(#.*)', pythonCode):  
>>> print(i.group())  
#number1  
#number2  
#Compute addition of two numbers
```

Pattern Matching (Cont.)

- regular expression to extract all the multi-line comments

```
""".*?"""
```

Note: a dot in a regular expression includes any character except end of line. However, if we include `re.DOTALL` as the third argument in the function **finditer**, dot matches newline character also. For example:

```
>>> for i in re.finditer(r'(""".*?""")', pythonCode, re.DOTALL):  
>>> print(i.group())
```

```
"""
```

Python code to add two numbers.

```
"""
```

Pattern Matching (Cont.)

```
>>> re.split(r',', 'Mira, Ronit, Vivek')  
['Mira', 'Ronit', 'Vivek']
```

`re.split ()`: returns a list of the substrings delimited by the regular expression provided

```
>>> re.split(r',|\n', '''Mira,Rohit,Vivek  
Aiysha,Renuka,Robin  
Sneha,Ravi''')  
['Mira', 'Rohit', 'Vivek', 'Aiysha', 'Renuka', 'Robin', 'Sneha', 'Ravi']
```

References

- [1] Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?