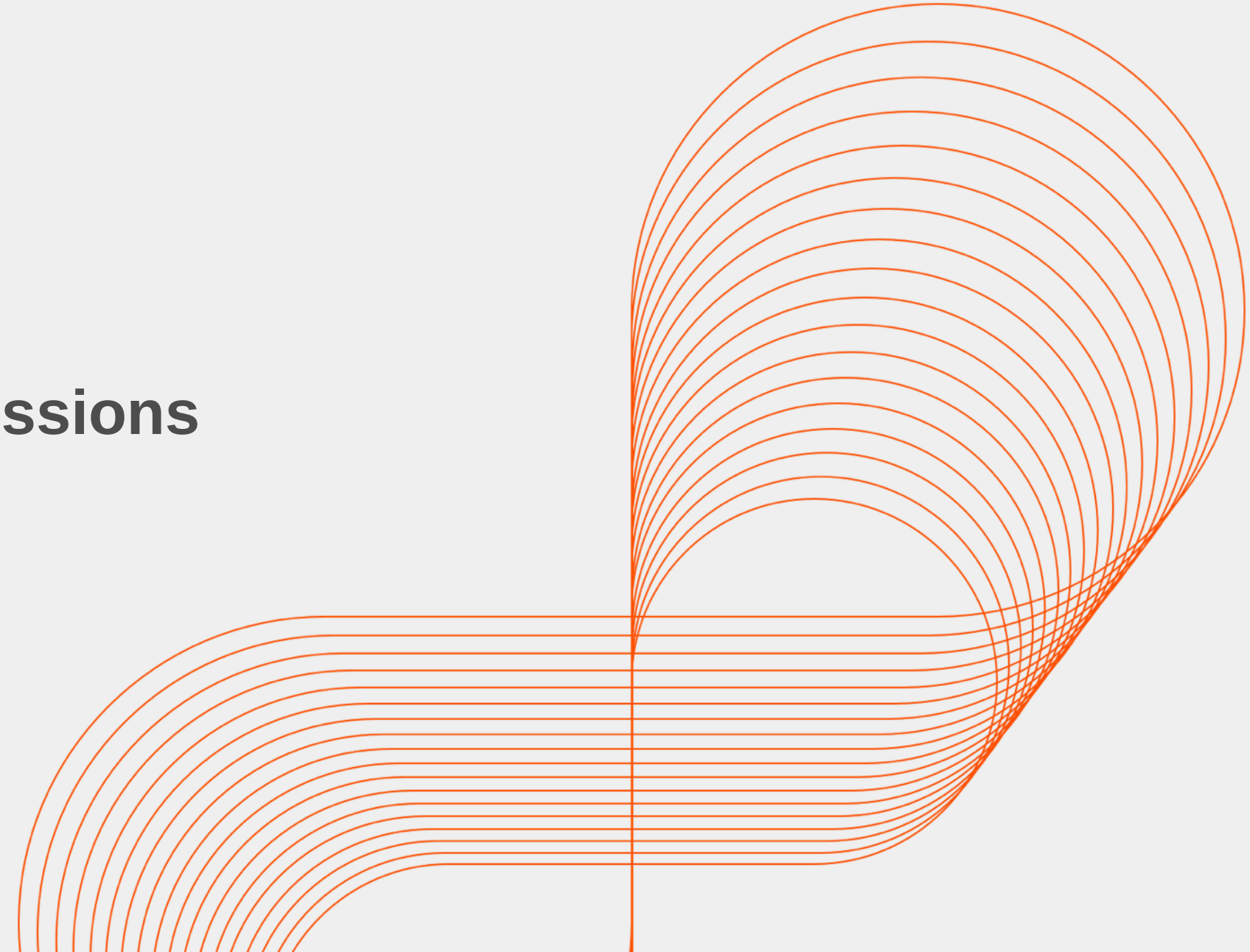
Persistent

Regular expressions



Objectives

At the end of this session, you will be able to understand:

- Regular Expressions in Python

Persistent

Regular Expression: re Module

Key learning points

Python Regular Expressions

- What are regular expressions?
- Searching Strings
- Regular Expressions
- Compiling Regular Expressions
- Regular Expression Repetition and Placement Characters
- Regular Expression String—Manipulation Functions
- Grouping

Persistent

Introduction to regular expressions

Regular Expressions (REs) provides an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality.

REs are simply strings which use special symbols and characters to indicate pattern repetition or to represent multiple characters so that they can "match" a set of strings with similar characteristics described by the pattern.

In other words, they enable matching of multiple strings.

Python supports REs through the standard library re module.

The re module provides regular expression tools for advanced string processing

Searching strings

- A Regular Expression (RE) is a way of recognizing and often extracting data from certain patterns of text.
- Python supports strings as a basic data type. Strings in Python are immutable sequences. Strings cannot be changed after they are created.
- In many applications, it is necessary to search for a character or set of characters in a string.
- For example, a programmer creating a word processor would want to provide capabilities for searching through documents.
- To perform such tasks, Python provides methods such as find and index.
- When searching for a substring, it can be determined whether a string contains the substring or can retrieve the index at which a substring begins.
- A regular expression that recognizes a piece of text or a string is said to match that text or string. An RE is defined by a string in which certain of the characters (the so-called metacharacters) can have a special meaning, which enables a single RE to match many different specific strings.

Special symbols and characters for regular expressions

The following table shows the most used metacharacters, special characters and symbols that give a developer the power and flexibility to use regular expressions.

Pattern	Description	Example RE
re_string	match literal string value **re_string**	hello
re1|re2	match literal string value **re1 or re2**	hello|Hello
f.O	match **any character (except NEWLINE)**	fao,f9o
^	match **start of string**	^Dear
\w	match any **alphanumeric character**, same as [A-Za-z0-9_] (\W is inverse of \w)	[A-Za-z_]\w+
$	match **end of string**	/bin/\w*sh$
*	match **0 or more occurrences of preceding RE**	[A-Za-z]\w*

Persistent

Special symbols and characters for regular expressions (contd.)

Pattern	Description	Example RE
\d	match any decimal **digit, same as [0–9] (\D** is inverse of \d: do not match any numeric digit)	data\d+.txt
+	match **1 or more occurrences of preceding RE**	\d+\.|\.\d+
?	match **0 or 1 occurrence(s) of preceding RE**	hello?
{N}	match **N occurrences of preceding RE**	\d{3}
{M,N}	match from **M to N occurrences of preceding RE**	\d{5,9}
[…]	match any single character from **character class**	[aeiou]
[..x–y..]	match any single character in the **range from x to y**	[0–9], [A-Za-z]
[^…]	**do not match any character from character class,** including any ranges, if present	[^aeiou]

Persistent

Special symbols and characters for regular expressions (contd.)

Pattern	Description	Example RE
(*|+|?|{})?	apply non-greedy versions of above occurrence/repetition symbols (*, +, ?, {})	.*?\w
(…)	match enclosed RE and save as **subgroup**	(\d{3})?, f(oo|u)bar
\s	match **any whitespace character, same as** [\n\t\r\v\f] (\S is inverse of \s)	of\sthe

Persistent

Matching more than one RE pattern with alternation

- The pipe symbol (|), a vertical bar on your keyboard, indicates an alternation operation, meaning that it is used to choose from one of the different regular expressions which are separated by the pipe symbol

For example, below are some patterns which employ alternation, along with the strings they match:

Pattern	Strings Matched
at|home	at, home
r2d2|c3po	r2d2, c3po
bat|bet|bit	bat, bet, bit

Matching any single character with dot symbol

- The dot or period (.) symbol matches any single character except for NEWLINE. Whether letter, number, whitespace not including "\n," printable, non-printable, or a symbol, the dot can match them all.
- Python REs have a compilation flag [S or DOTALL] which can override this to include NEWLINEs.
- In order to specify a dot character explicitly, its functionality needs to be escaped with a backslash, as in "\."

For example, below are some patterns which employ alternation, along with the strings they match:

Pattern	Strings Matched
f.o	any character between "f" and "o," e.g., fao,f9o, f#o, etc.
..	any pair of characters
.end	any character before the string end

Matching from the beginning or end of strings

- There are also symbols and related special characters to specify searching for patterns at the beginning and ending of strings. To match a pattern starting from the beginning, the carat symbol (^) must be used.
- Similarly, the dollar sign ($) will match a pattern from the end of a string

For example, below are some patterns which match from the beginning or end of strings:

Pattern	Strings Matched
^From	any string which starts with From
/bin/tcsh$	any string which ends with /bin/tcsh
^Subject: hi$	any string consisting solely of the string Subject: hi

Creating character classes ([ ])

- While the dot is good for allowing matches of any symbols, there may be occasions where there are specific characters that need to match. For this reason, the bracket symbols ([ ]) were invented.
- The regular expression will match from any of the enclosed characters.

Here are some examples:

Pattern	Strings Matched
b[aeiu]t	bat, bet, bit, but
[cr][23][dp][o2]	"r" or "c" then "2" or "3" followed by "d" or "p" and finally, either "o" or "2," Example c2do, r3p2, r2d2, c3po.

Multiple occurrence using closure operators

- The special symbols *, +, and ?, all of which can be used to match single, multiple, or no occurrences of string patterns. The asterisk or star operator (*) will match zero or more occurrences of the RE immediately to its left.
- The plus operator (+) will match one or more occurrences of an RE and the question mark operator (?) will match exactly 0 or 1 occurrences of an RE.
- There are also brace operators ({ }) with either a single value or a comma-separated pair of values. These indicate a match of exactly N occurrences (for {N}) or a range of occurrences, i.e., {M, N} will match from M to N occurrences. These symbols may also be escaped with the backslash, i.e., "*" matches the asterisk, etc.
- Finally, the question mark (?) is overloaded so that if it follows any of the following symbols, it will direct the regular expression engine to match as few repetitions as possible.

Multiple occurrence using closure operators (contd.)

Below are some examples:

Pattern	Strings Matched
[dn]ot?	"d" or "n," followed by an "o" and, at most, one "t" after that, i.e., do, no, dot, not.
0?[1–9]	any numeric digit, possibly prefixed with a “0,“ – example, the set of numeric representations of the months January to September, whether single- or double-digits.
[0–9]{15,16}	fifteen or sixteen digits, e.g., credit card numbers.
</?[^>]+>	strings which match all valid (and invalid) HTML tags.

Python re module functions

Python supports regular expressions through the re module.

The following table in this and the following slide lists the commonly used functions from the re module:

Function	Description
compile**(pattern, flags=0)**	compile RE **pattern with any optional flags and return** a regex object
match**(pattern, string, flags=0)**	attempt to match RE **pattern to string at the beginning with optional flags; return match object on success, none on failure**
search**(pattern, string, flags=0)**	search for first occurrence of **RE pattern within string** with optional **flags; return match object on success,** none on failure
findall**(pattern, string)**	look for all (non-overlapping) occurrences of **pattern in string; return a list of matches**

Python re module functions (contd.)

Pattern	Strings Matched
split**(pattern, string, max=0)**	split **string into a list according to RE pattern** delimiter and return list of successful matches, splitting at most **max times (split all occurrences is the default)**
group(num=0)	return entire match (or specific subgroup **num**)
groups()	return all matching subgroups in a tuple

Compiling REs with compile() function

- The following example shows how the re module’s compile function can be used to compile a regular expression into a compiled regular expression.
- This isn’t strictly necessary but compiled regular expressions can significantly increase a program’s speed, so they’re almost always used in programs that process large amounts of text.

Example:

```
import re

regexp = re.compile("hello")

count = 0

file = open("textfile", 'r’)

for line in file.readlines(): if regexp.search(line): count = count + 1

file.close()

print(count)
```

Compiling REs with compile() function (contd.)

- The program starts by importing the Python regular expression module, called re.
- It then takes the text string "hello" as a textual regular expression and compiles it into a compiled regular expression, using the re.compile function.
- What can the regular expression compiled from "hello" be used for? It can be used to recognize other instances of the word "hello" within another string; in other words, it can be used to determine whether another string contains "hello" as a substring.
- This is accomplished by the search method, which returns None if the regular expression isn’t found in the string argument.

Matching strings with match()

- The match() function attempts to match the pattern to the string, starting at the beginning.
- If the match is successful, a match object is returned, but on failure, None is returned. The group() method of a match object can be used to show the successful match.

Example:

```
>>> m = re.match(‘hello', ‘hello’)        # pattern matches string
>>> if m != None:                         # show match if successful
…         m.group()
`hello’
```

- The pattern “hello" matches exactly the string “hello.“ This can be confirmed that m is an example of a match object from within the interactive interpreter as shown below:

```
>>> m # confirm match object returned
<re.MatchObject instance at 80ebf48>
```


Persistent

Looking for a pattern within a string with search()

- The chances are greater that the pattern you seek is somewhere in the middle of a string, rather than at the beginning.
- This is where search() comes in handy. It works exactly in the same way as match except that it searches for the first occurrence of the given RE pattern anywhere with its string argument.
- Again, a match object is returned on success and None otherwise.

The following example illustrates the difference between match() and search()

```
>>> m = re.match('world', 'helloworld')   # no match
>>> if m != None: m.group()
```

As shown above there is no match here.

match() attempts to match the pattern to the string from the beginning, i.e., the "w" in the pattern is matched against the "h" in the string, which fails immediately.

However, the string "world" does appear elsewhere in "helloworld".

Looking for a pattern within a string with search() (contd.)

- The answer to the problem is by using the search() function.
- Rather than attempting a match, search() looks for the first occurrence of the pattern within the string.
- search() searches strictly from left to right.

Example:

```
>>> m = re.search('world', 'helloworld')
# use search() instead

>>> if m != None: m.group()

…

'world'      # search succeeds where match failed
```


Persistent

Finding every occurrence with findall()

- The findall() function looks for all non-overlapping occurrences of an RE pattern in a string
- It is similar to search() in that it performs a string search
- It differs from match() and search() in that findall() always returns a list
- The list will be empty if no occurrences are found but if successful, it will consist of all matches found, grouped in left-to-right order of occurrence

Example:

```
>>> re.findall('car', 'car')
['car']
>>> re.findall('car', 'scary')
['car']
>>> re.findall('car', 'carry the tarcardi to the car')
['car', 'car', 'car']
```


Persistent

Splitting on delimiting pattern with split()

- The split() function works similar to its string counterpart, but rather than splitting on a fixed string, it splits a string based on an RE pattern, adding some significant power to string splitting capabilities.
- If the delimiter given is not a regular expression which uses special symbols to match multiple patterns, then re.split() works in exactly the same manner as string.split() function.

The following example shows how split() function can be used to split a string on a single colon:

```
>>> re.split(':', 'str1:str2:str3')
['str1', 'str2', 'str3']
```


Persistent

Substituting text with regular expressions

- In addition to extracting strings from text, Python’s regular expression module can be used to find strings in text and substitute other strings in place of those that were found.
- This is accomplished using the regular substitution method sub.

The following example replaces instances of "the the" presumably a typo with single instances of "the“:

```
>>> import re
>>> string = "If the the problem is textual, use the the re module"
>>> pattern = r"the the"
>>> regexp = re.compile(pattern)
>>> regexp.sub("the", string)
'If the problem is textual, use the re module'
```

Assignments

1. Create a text file “emails.txt” and store a big list of valid and invalid email addresses on separate lines. Write a program to match the set of all valid e-mail addresses.

Summary

With this we have come to the end of our session, where we discussed about:

- Regular Expressions

In the next session we will discuss about

- Python OOP

Persistent

Reference material

- http://www.tutorialspoint.com/python
- http://www.learnpython.org/
- http://docs.python.org/2/tutorial/
- https://docs.python.org/2/tutorial/stdlib.html
- https://docs.python.org/2/tutorial/stdlib2.html

Persistent

Questions

Key contacts

Sakshi Jamgaonkar

sakshi_jamgaonkar@persistent.com

Asif Immanad

asif_immanad@persistent.co.in

Persistent

Persistent

Thank you!