Unit-III

By:
Mani Butwall,
Asst. Prof. (CSE)

Mani Butwall (CSE), ITM University

Contents

- Regular Expressions: Introduction/Motivation
- Special Symbols and Characters for Res
- REs and Python
- Handling regular expressions and their built-in methods
- Lambda Expression
- Map()
- Reduce()
- Filter()

Regular Expression

- A **Regular Expression (RE)** in a programming language is a special text string used for describing a search pattern.
- It is extremely useful for extracting information from text such as code, files, log, spreadsheets or even documents.
- While using the Python regular expression the first thing is to recognize is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters also referred as string.
- Ascii or latin letters are those that are on your keyboards and Unicode is used to match the foreign text.
- It includes digits and punctuation and all special characters like $#@!%, etc.

- Regular expression or RegEx in Python is denoted as RE (REs, regexes or regex pattern) are imported through **re module**.
- Python supports regular expression through libraries. RegEx in Python supports various things like **Modifiers, Identifiers, and White space characters**.

- **“A Regular Expression is used for identifying a search pattern in a text string. It also helps in finding out the correctness of the data and even operations such as finding, replacing and formatting the data is possible using Regular Expressions.”**

Why Use Regular Expression?

- To answer this question, we will look at the various problems faced by us which in turn is solved by using Regular Expressions.
- Consider the following scenario:
- You have a log file which contains a large sum of data. And from this log file, you wish to fetch only the date and time. As you can look at the image, readability of the log file is low upon first glance.
- Regular Expressions can be used in this case to recognize the patterns and extract the required information easily.

LOG

Jul 1 03:26:12 syslog: [m_java][1/Jul/2013 03:27:12.818][j:[SessionThread<]^latcom/avc/abc/magr/service/find.something(abc/1235/locator/abc;Ljava/lang/String;)Labc/abc/abcd/abcd;(bytecode:7)

Example 2

- Consider the next scenario – You are a salesperson and you have a lot of email addresses/ mobile numbers and a lot of those addresses are fake/invalid.

Phone Number
444-122-1234
123-122-78999
111-123-23
67-7890-2019

- How do we verify the phone number and then classify it based on the country of origin?
- Every correct number will have a particular pattern which can be traced and followed through by using Regular Expressions.

Example 3

- Next up is another simple scenario:

- We have a Student Database containing details such as name, age, and address. Consider the case where the Area code was originally **59006** but now has been changed to **59076.** To manually update this for each student would be time-consuming and a very lengthy process.
- Basically, to solve these using Regular Expressions, we first find a particular string from the student data containing the pin code and later replace all of them with the new ones.

- Regular expressions can be used with multiple languages. Such as:

1. Java
2. Python
3. Ruby
4. Swift
5. Scala
6. Groovy
7. C#
8. PHP
9. Javascript

RegEx Function

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

```
In [42]: import re
    ...:
    ...: pattern="Ma"
    ...: data=input("Enter string : ")
    ...:
    ...: result=re.match(pattern,data)
    ...:
    ...: if result:
    ...:     print("Successfull")
    ...: else:
    ...:     print("not successfull")
    ...:
    ...:

Enter string : Mani
Successfull
```

```
In [44]: import re
    ...:
    ...: pattern="Ma"
    ...: data=input("Enter string : ")
    ...:
    ...: result=re.match(pattern,data)
    ...:
    ...: if result:
    ...:     print("Successfull")
    ...: else:
    ...:     print("not successfull")
    ...:
    ...:

Enter string : Mnemonics
not successfull
```

```
In [46]: import re
    ...:
    ...: pattern="Ma"
    ...: data=input("Enter string : ")
    ...:
    ...: result=re.search(pattern,data)
    ...:
    ...: if result:
    ...:     print("Successfull")
    ...: else:
    ...:     print("not successfull")
    ...:
    ...:

Enter string : NMani
Successfull
```

```
In [47]: import re
    ...:
    ...: pattern="Ma"
    ...: data=input("Enter string : ")
    ...:
    ...: result=re.match(pattern,data)
    ...:
    ...: if result:
    ...:     print("Successfull")
    ...: else:
    ...:     print("not successfull")
    ...:
    ...:

Enter string : NMani
not successfull
```

Metacharacters

- Metacharacters are characters with a special meaning:

\	Signals a special sequence (can also be used to escape special characters)	"\d"
.	Any character (except newline character)	"he..o"
^	Starts with	"^hello"
$	Ends with	"world$"
*	Zero or more occurrences	"aix*"
[]	A set of characters	"[a-m]"

{}	Exactly the specified number of occurrences	"al{2}"
|	Either or	"falls|stays"
()	Capture and group	

Examples

`[]` - Square brackets

Square brackets specifies a set of characters you wish to match.

Expression	String	Matched?
`[abc]`	`a`	1 match
	`ac`	2 matches
	`Hey Jude`	No match
	`abc de ca`	5 matches

Here, `[abc]` will match if the string you are trying to match contains any of the `a`, `b` or `c`.

You can also specify a range of characters using `-` inside square brackets.

Mani Butwall (CSE), ITM University

You can also specify a range of characters using `-` inside square brackets.

- `[a-e]` is the same as `[abcde]`.
- `[1-4]` is the same as `[1234]`.
- `[0-39]` is the same as `[01239]`.

You can complement (invert) the character set by using caret `^` symbol at the start of a square-bracket.

- `[^abc]` means any character except `a` or `b` or `c`.
- `[^0-9]` means any non-digit character.

`.` - Period

A period matches any single character (except newline `'\n'`).

Expression	String	Matched?
`..`	`a`	No match
	`ac`	1 match
	`acd`	1 match
	`acde`	2 matches (contains 4 characters)

`^` - Caret

The caret symbol `^` is used to check if a string **starts with** a certain character.

Expression	String	Matched?
`^a`	`a`	1 match
	`abc`	1 match
	`bac`	No match
`^ab`	`abc`	1 match
	`acb`	No match (starts with `a` but not followed by `b`)

`$` - Dollar

The dollar symbol `$` is used to check if a string **ends with** a certain character.

Expression	String	Matched?
`a$`	`a`	1 match
	`formula`	1 match
	`cab`	No match

`*` - Star

The star symbol `*` matches **zero or more occurrences** of the pattern left to it.

Expression	String	Matched?
`ma*n`	`mn`	1 match
	`man`	1 match
	`maaan`	1 match
	`main`	No match (`a` is not followed by `n`)
	`woman`	1 match

`+` - Plus

The plus symbol `+` matches **one or more occurrences** of the pattern left to it.

Expression	String	Matched?
`ma+n`	`mn`	No match (no `a` character)
	`man`	1 match
	`maaan`	1 match
	`main`	No match (a is not followed by n)
	`woman`	1 match

`?` - Question Mark

The question mark symbol `?` matches **zero or one occurrence** of the pattern left to it.

Expression	String	Matched?
`ma?n`	`mn`	1 match
	`man`	1 match
	`maaan`	No match (more than one `a` character)
	`main`	No match (a is not followed by n)
	`woman`	1 match

`{}` - Braces

Consider this code: `{n,m}`. This means at least `n`, and at most `m` repetitions of the pattern left to it.

Expression	String	Matched?
`a{2,3}`	`abc dat`	No match
	`abc daat`	1 match (at `daat`)
	`aabc daaat`	2 matches (at `aabc` and `daaat`)
	`aabc daaaat`	2 matches (at `aabc` and `daaaat`)

Let's try one more example. This RegEx `[0-9]{2, 4}` matches at least 2 digits but not more than 4 digits

Expression	String	Matched?
`[0-9]{2,4}`	`ab123csde`	1 match (match at `ab123csde`)
	`12 and 345673`	3 matches (`12`, `3456`, `73`)
	`1 and 2`	No match

| - Alternation

Vertical bar | is used for alternation (or operator).

Expression	String	Matched?
a|b	cde	No match
	ade	1 match (match at ade)
	acdbea	3 matches (at acdbea)

Here, a|b match any string that contains either a or b

`()` - Group

Parentheses `()` is used to group sub-patterns. For example, `(a|b|c)xz` match any string that matches either `a` or `b` or `c` followed by `xz`

Expression	String	Matched?
`(a|b|c)xz`	`ab xz`	No match
	`abxz`	1 match (match at `abxz`)
	`axz cabxz`	2 matches (at `axzbc cabxz`)

`\` - Backslash

Backlash `\` is used to escape various characters including all metacharacters. For example,

`\$a` match if a string contains `$` followed by `a`. Here, `$` is not interpreted by a RegEx engine in a special way.

If you are unsure if a character has special meaning or not, you can put `\` in front of it. This makes sure the character is not treated in a special way.

Special Sequences

- A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

Character	Description	Example
\A	Returns a match if the specified characters are at the beginning of the string	"\AThe"
\b	Returns a match where the specified characters are at the beginning or at the end of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\bain" r"ain\b"
\B	Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word (the "r" in the beginning is making sure that the string is being treated as a "raw string")	r"\Bain" r"ain\B"

\d	Returns a match where the string contains digits (numbers from 0-9)	"\d"
\D	Returns a match where the string DOES NOT contain digits	"\D"
\s	Returns a match where the string contains a white space character	"\s"
\S	Returns a match where the string DOES NOT contain a white space character	"\S"

\w	Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character)	"\w"
\W	Returns a match where the string DOES NOT contain any word characters	"\W"
\Z	Returns a match if the specified characters are at the end of the string	"Spain\Z"

Examples

`\A` - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
`\Athe`	`the sun`	Match
	`In the sun`	No match

`\b` - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
`\bfoo`	`football`	Match
	`a football`	Match
	`afootball`	No match
`foo\b`	`the foo`	Match
	`the afoo test`	Match
	`the afootest`	No match

`\B` - Opposite of `\b`. Matches if the specified characters are **not** at the beginning or end of a word.

Expression	String	Matched?
`\Bfoo`	`football`	No match
	`a football`	No match
	`afootball`	Match
`foo\B`	`the foo`	No match
	`the afoo test`	No match
	`the afootest`	Match

`\d` - Matches any decimal digit. Equivalent to `[0-9]`

Expression	String	Matched?
`\d`	`12abc3`	3 matches (at `12abc3`)
	`Python`	No match

`\D` - Matches any non-decimal digit. Equivalent to `[^0-9]`

Expression	String	Matched?
`\D`	`1ab34"50`	3 matches (at `1ab34"50`)
	`1345`	No match

`\s` - Matches where a string contains any whitespace character. Equivalent to `[ \t\n\r\f\v]`.

Expression	String	Matched?
`\s`	`Python RegEx`	1 match
	`PythonRegEx`	No match

`\S` - Matches where a string contains any non-whitespace character. Equivalent to `[^ \t\n\r\f\v]`.

Expression	String	Matched?
`\S`	`a b`	2 matches (at `a b`)
		No match

`\w` - Matches any alphanumeric character (digits and alphabets). Equivalent to `[a-zA-Z0-9_]`. By the way, underscore `_` is also considered an alphanumeric character.

Expression	String	Matched?
`\w`	`12&": ;c`	3 matches (at `12&": ;c`)
	`%"> !`	No match

`\W` - Matches any non-alphanumeric character. Equivalent to `[^a-zA-Z0-9_]`

Expression	String	Matched?
`\W`	`1a2%c`	1 match (at `1a2%c`)
	`Python`	No match

`\Z` - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
`Python\Z`	`I like Python`	1 match
	`I like Python Programming`	No match
	`Python is fun.`	No match

Set

- A set is a set of characters inside a pair of square brackets [] with a special meaning:

Set	Description
[arn]	Returns a match where one of the specified characters (a, r, or n) are present
[a-n]	Returns a match for any lower case character, alphabetically between a and n
[^arn]	Returns a match for any character EXCEPT a, r, and n
[0123]	Returns a match where any of the specified digits (0, 1, 2, or 3) are present
[0-9]	Returns a match for any digit between 0 and 9

[0-5][0-9]	Returns a match for any two-digit numbers from 00 and 59
[a-zA-Z]	Returns a match for any character alphabetically between a and z, lower case OR upper case
[+]	In sets, +, *, ., |, (), $,{} has no special meaning, so [+] means: return a match for any + character in the string

Compile()

- Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
In [103]: import re
     ...:
     ...: p=re.compile('[a-h]')
     ...:
     ...: r=p.findall("Ashi Hello This is Python Class")
     ...: print(r)
     ...:
['h', 'e', 'h', 'h', 'a']
```

- compile() creates regular expression character class [a-h],

which is equivalent to [abcdefgh].

class [abcdefgh] will match with string with 'a', 'b', 'c', 'd', 'e‘ and so on.

Findall()

- The re.findall() method returns a list of strings containing all matches.

```
# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

Mani Butwall (CSE), ITM University

Split()

- Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

- **Syntax :**
- re.split(pattern, string, maxsplit=0, flags=0)

- The First parameter, pattern denotes the regular expression, string is the given string in which pattern will be searched for and in which splitting occurs, maxsplit if not provided is considered to be zero ‘0’, and
- if any nonzero value is provided, then at most that many splits occurs. If maxsplit = 1, then the string will split once only, resulting in a list of length 2. The flags are very useful and can help to shorten code, they are not necessary parameters, eg: flags = re.IGNORECASE, In this split, case will be ignored.

```
import re
p="\s+"
d="Hii 101, u scored 345"

r=re.split(p,d)
print(r)
```

```
import re
p=","
d="Hello Python This is my first prpgram, Mani,Shikha,Sonam"

r=re.split(p,d)
print(r)
```

```
import re

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

```
import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'

# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

Sub()

The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter), and upon finding the substring pattern is replaced by repl(2nd parameter), count checks and maintains the number of times this occurs.

- **Syntax:**
- re.sub(pattern, repl, string, count=0, flags=0)

```
In [75]: import re
    ...: p="py"
    ...: rpl="**"
    ...: d="You can copy python program from soupy doupy"
    ...:
    ...: r=re.sub(p,rpl,d,count=0)
    ...: print(r)
You can co** **thon program from sou** dou**
```

Subn()

- subn() is similar to sub() in all ways, except in its way to providing output. It returns a tuple with count of total of replacement and the new string rather than just the string.

- **Syntax:**
- re.subn(pattern, repl, string, count=0, flags=0)

```
In [79]: import re
    ...: p="py"
    ...: rpl="**"
    ...: d="You can copy python program from soupy doupy"
    ...:
    ...: r=re.subn(p,rpl,d)
    ...: print(r)
('You can co** **thon program from sou** dou**', 4)
```

Search()

- The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.
- If the search is successful, re.search() returns a match object; if not, it returns None.
- match = re.search(pattern, str)

```
import re

string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
  print("pattern found inside the string")
else:
  print("pattern not found")

# Output: pattern found inside the string
```

Group()

- The group() method returns the part of the string where there is a match.

```
In [89]: import re
    ...: p="(\w+)#(\d+)"
    ...: d="mani#1234"
    ...:
    ...: r=re.match(p,d)
    ...:
    ...: print(r.group())
    ...: print(r.group(1))
    ...: print(r.group(2))
    ...:
mani#1234
mani
1234
```

```
In [91]: import re
    ...: p="(\w+)/(\w+)/(\w+)/(\w+)"
    ...: d="C/programfile/bin/jre"
    ...:
    ...: r=re.match(p,d)
    ...: print(r.group())
    ...: print(r.group(1))
    ...: print(r.group(2))
    ...: print(r.group(3))
    ...: print(r.group(4))
C/programfile/bin/jre
C
programfile
bin
jre
```

match.start(), match.end() and match.span()

- The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.
- >>> **match.start()**
- 2
- >>> **match.end()**
- 8
- The span()
- function returns a tuple containing start and end index of the matched part.
- >>> **match.span()**
- (2, 8)

Escape()

- Return string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

- **Syntax:**
- re.escape(string)

```
In [80]: import re
    ...: d="Hello 101 u scored 345 @12#$45$78&90"
    ...:
    ...: r=re.escape(d)
    ...: print(r)
Hello\ 101\ u\ scored\ 345\ @12\#\$45\$78\&90
```

Summary

- A regular expression in a programming language is a special text string used for describing a search pattern. It includes digits and punctuation and all special characters like $#@!%, etc. Expression can include literal

1. Text matching
2. Repetition
3. Branching
4. Pattern-composition etc.

- In Python, a regular expression is denoted as RE (REs, regexes or regex pattern) are embedded through Python re module.

- "re" module included with Python primarily used for string searching and manipulation
- Also used frequently for webpage "Scraping" (extract large amount of data from websites)
- **Regular Expression Methods include** re.match(),re.search()& re.findall()
- Other Python RegEx replace methods are sub() and subn() which are used to replace matching strings in re
- **Python Flags** Many Python Regex Methods and Regex functions take an optional argument called Flags
- This flags can modify the meaning of the given Regex pattern
- Various Python flags used in Regex Methods are re.M, re.I, re.S, etc.

Lambda Expression

- A **Lambda Function in Python** programming is an anonymous function or a function having no name. It is a small and restricted function having no more than one line. Just like a normal function, a Lambda function can have multiple arguments with one expression.
- In Python, lambda expressions (or lambda forms) are utilized to construct anonymous functions. To do so, you will use the **lambda** keyword (just as you use **def** to define normal functions). Every anonymous function you define in Python will have 3 essential parts:

1. The lambda keyword.
2. The parameters (or bound variables), and
3. The function body.

- A lambda function can have any number of parameters, but the function body can only contain **one** expression. Moreover, a lambda is written in a single line of code and can also be invoked immediately.

Syntax of Lambda Expresiion

- **Syntax:** lambda arguments: expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Filter()

- The filter function is used to select some particular elements from a sequence of elements. The sequence can be any iterator like lists, sets, tuples, etc.
- The elements which will be selected is based on some pre-defined constraint. It takes 2 parameters:

1. A function that defines the filtering constraint
2. A sequence (any iterator like lists, tuples, etc.)

```
l = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

newlist = list(filter(lambda x: (x%2 != 0) , li))
print(newlist)
```

```
ages = [13, 90, 17, 59, 21, 60, 5]

adults = list(filter(lambda age: age>18, ages))

print(adults)
```

Map()

- The map function is used to apply a particular operation to every element in a sequence. Like filter(), it also takes 2 parameters:

1. A function that defines the op to perform on the elements
2. One or more sequences

```
In [5]: names = ['mani', 'shikha', 'sonam', 'shuchi']
   ...: newlist= list(map(lambda names: str.upper(names), names))
   ...: print(newlist)
['MANI', 'SHIKHA', 'SONAM', 'SHUCHI']

In [6]: li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
   ...: final_list = list(map(lambda x: x*2, li))
   ...: print(final_list)
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Reduce()

- The reduce function, like map(), is used to apply an operation to every element in a sequence. However, it differs from the map in its working. These are the steps followed by the reduce() function to compute an output:
- **Step 1)** Perform the defined operation on the first 2 elements of the sequence.
- **Step 2)** Save this result
- **Step 3)** Perform the operation with the saved result and the next element in the sequence.
- **Step 4)** Repeat until no more elements are left.
- It also takes two parameters:

1. A function that defines the operation to be performed
2. A sequence (any iterator like lists, tuples, etc.)

Example

- reduce() is a bit harder to understand than map() and filter(), so let's look at a step by step example:

1. We start with a list [2, 4, 7, 3] and pass the add(x, y) function to reduce() alongside this list, without an initial value
2. reduce() calls add(2, 4), and add() returns 6
3. reduce() calls add(6, 7) (result of the previous call to add() and the next element in the list as parameters), and add() returns 13
4. reduce() calls add(13, 3), and add() returns 16
5. Since no more elements are left in the sequence, reduce() returns 16

```
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

```
from functools import reduce
sequences = [1,2,3,4,5]
product = reduce (lambda x, y: x*y, sequences)
print(product)
```

```
In [11]: from functools import reduce
    ...: l=[5,6,7,8,9,3,4,2]
    ...: output=reduce(lambda a,b:a if a>b else b,l)
    ...: print(output)
9
```

Why (and why not) use lambda functions?

- Lambdas provide compact syntax for writing functions which return a single expression.
- However, you should know when it is a good idea to use lambdas and when to avoid them.
- One of the most common use cases for lambdas is in functional programming as Python supports a paradigm (or style) of programming known as functional programming.
- It allows you to provide a function as a parameter to another function (for example, in map, filter, etc.). In such cases, using lambdas offer an elegant way to create a one-time function and pass it as the parameter.

When should you not use Lambda?

- You should never write complicated lambda functions in a production environment. It will be very difficult for coders who maintain your code to decrypt it.
- If you find yourself making complex one-liner expressions, it would be a much superior practice to define a proper function.
- As a best practice, you need to remember that simple code is always better than complex code.

Lambdas vs. Regular functions

- As previously stated, lambdas are just functions which do not have an identifier bound to them.
- In simpler words, they are functions with no names (hence, anonymous).

1. Lambda functions can only have one expression in their body.
2. Regular functions can have multiple expressions and statements in their body.
3. Lambdas do not have a name associated with them. That's why they are also known as anonymous functions.
4. Regular functions must have a name and signature.
5. Lambdas do not contain a return statement because the body is automatically returned.
6. Functions which need to return value should include a return statement.

Summary

- Lambdas, also known as anonymous functions, are small, restricted functions which do not need a name (i.e., an identifier).
- Every lambda function in Python has 3 essential parts:

1. The lambda keyword.
2. The parameters (or bound variables), and
3. The function body.

- The syntax for writing a lambda is: lambda parameter: expression
- Lambdas can have any number of parameters, but they are not enclosed in braces
- A lambda can have only 1 expression in its function body, which is returned by default.

- At the bytecode level, there is not much difference between how lambdas and regular functions are handled by the interpreter.
- Lambdas support IIFE thru this syntax: (lambda parameter: expression)(argument)
- Lambdas are commonly used with the following python built-ins:
- Filter: filter (lambda parameter: expression, iterable-sequence)
- Map: map (lambda parameter: expression, iterable-sequences)
- Reduce: reduce (lambda parameter1, parameter2: expression, iterable-sequence)
- Do not write complicated lambda functions in a production environment because it will be difficult for code-maintainers.