PYTHON REGULAR EXPRESSIONS

LEARN PYTHON REGULAR EXPRESSIONS IN AN EASY WAY

PRESENTED BY

ACODEMY

By Acodemy

PYTHON REGULAR EXPRESSIONS

The ultimate crash course to learn Python Regular Expressions FAST!

This book is written to guide those who are passionate to learn how to use Python Regular Expressions. It has been written to help those who intend to learn basic and complex tools of Python Regular Expressions.

Table of Contents

Regular Expressions

A brief overview

Repeating things

Alternation

Chapter 4: Quantifiers and Boundary Matchers

Quantifiers
Greedy as well as reluctant quantifiers Possessive Quantifiers

Boundary Matchers

Chapter 5: Special characters, module contents and Regular Expression Objects

Module Contents

Regular Expression Objects

Chapter 6: Grouping

Expressions with Python

Backreferences

Named groups

Atomic groups

Specialized cases with groups

Chapter 7: Look Around

Look Ahead

Look Behind

Look around and groups

Chapter 8: Measuring performance of regexes

Be specific

Creating a Phonebook

Data Munging

Discovering all Adverbs
Discovering all Adverbs and their Locations

Chapter 1: Introduction to Regular expressions

Chapter Objective:

The main objective of this chapter is to make the readers familiar with what a regular expression is. The chapter covers most of the basic concepts related to regular expressions. First of all, a basic explanation of regulation expressions is given and then the history so that the reader has an idea of how regular expressions evolved with time and their importance in the present scenario. The next part gives an overview of the most basic terms related to regular expressions like literal characters, character classes, back references, *etc.* In the end, an exercise to test what the reader has gained from the chapter has been given.

Regular expressions meaning

Regular expressions are also termed as regexes, REs or regex patterns. They are basically a minute, extremely specialized programming language implanted inside Python and the re module is used for making them available. By employing this tiny language, you specify the rules for the probable strings set which you wish to match. Such a set can contain e-mail addresses, English sentences, or TeX commands, or whatever you wish. You may then ask questions like “Is this string matching the

Pattern?” You may also employ Regular Expressions for modifying a string or splitting it apart in a variety of ways.

RE patterns have been compiled into a sequence of bytecodes. The execution of the byte code takes place by a matching engine that is written in C language. It might be required to give precise attention to how the engine will be executing a particular Regular Expression, and write the Regular Expression in a particular way for creating to bytecode which runs faster.

REs can be considered as text patters describing the form string of text should possess. A number of functions can be accomplished by employing regular expressions. The chief among them being the following:

Find out particular pattern appearance in a specific text. To exemplify, it is possible to examine whether a word “hard” is present in a document by employing only one scan.

Extraction of a particular part of a text. To exemplify, the extraction of a street number of any address.

Replacement of a particular part of text such as altering the color of a specific text to violet.

Breaking down a text in to chunks by employing a punctuation, *etc.*

History of regular expressions

Regular expressions gained fame in the late 1960s. The maiden appearances of Res in programming can be credited to develop Kleene's notation into the Quick Editor as a way to for matching various patterns in several text files. For speed, the implementation of regular expression was performed by using JIT to IBM 7094 code on the Compatible Time-Sharing System. Ken Thompson.

A comparatively more complex Res came into picture in Perl in the late 1980s. They were actually derived from a regex library. In the present scenario, Res are largely backed in text processing programs, programming languages and various other programs. Re back is an element of the standardized library of various programming languages which comprise Python, *etc.* and is created into the syntax of others which comprise Perl and ECMA Script. Employments of Re functionality are mostly known as a regular expression engine, and various libraries are there for reuse.

Res are universal. They are present in the most novice Javascript framework and can also be traced in the UNIX tools of the 1970s. Any modern programming language is incomplete if it does not back Res.

Though they are found in various languages, they are not found in the toolkit of the current coders. The difficult learning curve offered by them can be one of the causes of this. Being an expert in Res is not so easy and they are very complicated to read when not written appropriately.

Once you read this book, you will be able to learn the most competent practices while composing regular expressions to ease the reading procedure. Though Res are present in the newest and the most effective programming languages presently, their history can be traced back to 1943 when a logical calculus of the concepts existing in nervous activity was released by Warren McCulloch and Walter Pitts. This research was actually the starting point of the regular expressions and offered the maiden mathematical model of a neural network. In the year 1968, a renowned contributor of computer science (Ken Thompson) considered the findings of Kleene and further expanded it, publishing his works in the Regular Expression Search Algorithm titled paper. His contributions were not limited by just a single paper. He encompassed support for these Res in his version of Quick Editor.

Other remarkable milestones worth mentioning here are the issue of the maiden non-proprietary library of regex by Henry Spence and after that, the development of the Perl by famous computer science contributor, Larry Wall. The implementation in Perl was taken to another step and contributed a number of modifications to the actual RE syntax thereby making the famous Perl flavor. A number of the implementations that followed in other languages or tools are depended on the Perl flavor of Re.

The well-known Jargon file mentions them as regex, reg-ex and regexp. Though there is not a very authoritarian approach of naming Res, they are basically based on formal languages which imply that everything has to be appropriate and exact. A variety of the current implementations back features which cannot be represented in formal languages and hence they cannot be considered as real Res. All the pre-mentioned terms (regex, reg-ex and regexp, REs, Regular expressions will be used as synonyms in this book.

Most important concepts related to regular expressions

Literal Characters

The most basic regex comprises of a single literal character. It matches the first occurrence of that character in the string. The maiden occurrence of the string character is mapped by it. For example, consider a string Jam is Sweet. In this case, the ‘ a’ after the J will be matched by the regular expression.

Character classes

The “character class” is employed for telling the regex engine that it needs to match a single character among a group of characters. In order to match a character that you want to be matched, just put it inside square brackets. Suppose you wish to match an o or an e, you will use [oe]. It is extremely helpful when you are not sure if the document you are looking for has been created in British or English style.

Only one character is matched by it. For example, pr[ae]y will not match praay, praey or any similar characters. For a character class, it does not take into consideration what the order of the characters is and the outcome is the same. A hyphen can be employed in a character class for specifying a particular character set. [0-20] will provide the matching of a digit that ranges in between 0 to 9.

Character class is a feature of regular expressions that has the maximum usage. A word can be searched even though it is not spelled correctly like pr[ae]y.

Dot

The dot also known as period is a very well-known metacharacter. Unluckily, it has a very high chances of being misused. The function of the dot is to match a character not taking into consideration what is actually is. Only the line break characters are not matched by it.

Anchors

Anchors function by matching a particular position that lies either before or after or between characters instead of matching any specific character. They can be employed for anchoring the regular expression match at a specific location. Special character ^is used for matching a location that is just before the maiden character in the string. For example, if you apply ^d to dec, it matches d.

Word Boundaries

The \b is a metcharacter that is employed for matching a location termed as word boundary. Such a match is 0-length.
The following are the three different locations in order to be considered as word boundaries
· In case the first character is a word character, the location qualifying as a word boundary is before it
· In case the last character is a word character, the location qualifying as a word boundary is after it.
· The location of the word boundary is in between 2 string characters wherein the first is a word character whereas the second one is not so.

Alternation

Alternation is employed for matching a particular regular expression among a group of various probable regex. For example, if you wish to find the text mouse or fish, separate these two by using a pipe symbol. Among all the regular expression operators, alternation has the least priority. This means that it directs the regular expression engine that it has to match either to the right or left of the vertical bar. You must employ parenthesis in order to group in case you wish to limit the extent to which the alternation will reach.

Repetition operator

There are three main repetition operators: question mark, star and asterisk. The preceding token in the regex is made elective by using question mark. For example flavour will match both flavor and flavour. Hence a number of tokens can be made optional by placing them together with the help of parenthesis and putting the question mark. For example Sept(ember)? Will match Sept as well as September. The asterisk or star commands the regex engine to try to match the former token either zero or multiple times.

Using parenthesis

A portion of a regex can be **grouped** together by getting it placed inside parenthesis or round brackets. It will enable you to use a quantifier for the whole group or to limit alternation to a particular portion of the regular expression. For **grouping** only parenthesis can be put into use. **'[]'brackets** describe a character class whereas **'{}' brackets** are employed by a quantifier with particular limits.

Back references

They are employed for matching the text that has been earlier matched by a capturing group. Suppose you wish to match a pair of opening as well as closing HTML tags as well as the text in them. The tag's name can be used again by placing the opening tag into a back reference.

Named capturing groups:

Almost all the regex engines back numbered capturing groups. Lengthy regex with several groups back references are difficult to read. They might be specifically hard to maintain since the addition or elimination of a capturing group in the regular expression's middle disappoints the counts of all the groups following the added or the eliminated group.

The re module of python is the maiden one providing a solution in terms of named back references as well as named capturing groups. The group's match is captured by (?P<name>group) in the "name" back reference. It is necessary that the name should be an alphanumeric arrangement that begins with a letter whereas group could be any regex. The group's matter might be referenced with the named back reference that is (?P=name). All the three things inside the bracket are the syntax's part. Even though the named back references employ parenthesis, this is simply a back reference wherein no grouping is done.

Free spacing regex:

The whitespacing between regex tokens is ignored in the free-spacing mode. The Whitespace comprises of tabs, spaces as well as line breaks. d e f means the same as def in free-spacing mode, whereas \ d and \d cannot be considered as the same. The first one matches d whereas the second matches a digit. \d is a single regex token composed of a backslash and a "d". Breaking up the token with a space gives you an escaped space (which matches a space), and a literal "d". Only one regular expression token is represented by \d but splitting the token with a space offers you an extra space as well as a literal “d”.

Summary

Now that you have gone through this chapter, you should have able to know what a regular expression is how it came into picture and the basics related to it. To revise: Regular expressions are basically a minute, extremely specialized programming language implanted inside Python and the re module is used for making them available. They are also termed as regexes, REs or regex patterns. A literal character matches the first occurrence of that character in the string. The other important terms related to regular expressions are: special characters, dot, anchors, back references, word boundaries, *etc.* You must have been familiar with all these by now.

Assignment

Exercise 1

The execution of the byte code takes place by a matching engine. Choose the language in which this matching engine is written.

a) C

b) C++

c) Java d) None The correct answer is (a)

Exercise 2

Who among the following released the first non-proprietary library of regex?

a) Henry Space b) Larry Wall c) Ken Thompson d) None The correct answer is (a)

Exercise 3

By applying ^d to dec, which one among the following is matched?

a) e b) d c) c d) None The correct answer is (b)

Exercise 4

Which one among the following has the function of matching a character not taking into consideration what is actually is?

a) anchor b) dot The correct answer is (b)

Exercise 5

Nov and ember will be matched by using which one among the following?

a) Nov(ember) b) Nov(ember)?

The correct answer is (b)

Exercise 6

\ d will imply a) d b) a digit The correct answer is (a)

Chapter 2: Regex with Python

Chapter Objective: In the first chapter, we've had an overview of the basics related to regular expressions. In the second chapter you will have an understanding of the various operations offered by Python and how the regular expression is handled by python. To accomplish that, we will notice the twists of Python while handling Res, the various string types, the Application Program Interface it provides via the MatchObject and RegexObject classes, each task which can be achieved with them in detail and some issues generally encountered by users.

A brief overview As version 1.5, Python offers a Perl-style regex with a few intelligent exceptions which we encounter afterwards, patterns and strings that are to be found may be 8 bit or Unicode strings.

Tip
Unicode is employed to denote the living characters of the word as well as important scripts. It might be imagined as a calibration between code points and characters. Therefore any character can be denoted irrespective of its language with a single number.
Res are backed by the re module. Therefore, similar to other modules in Python, it is only required to import it to begin playing with them. In order to do that, you require to begin the Python interactive shell by employing this code: > > > import re After you have done this, you may start attempting to match any pattern. In order to accomplish that, you require the compilation of a pattern, converting it into bytecode, as described below. The execution of the bytecode takes place afterwards in an engine that is programmed in C programming language.
> > > pattern = re.compile(r'\ bfoo\ b') **Tip**
Bytecode is the output that is produced by programming languages that would be afterwards understood by an interpreter. It can be considered as an intermediate language. The most apt illustration is the Java bytecode interpreted by JVM. After the compilation of the pattern, it can be compared with a string, this is explained below: > > > pattern.match(" foo bar") < *sre.SRE*Match at 0x108acac60 > In the example given, the compilation of a pattern takes place and after that it is found if the matching of the pattern takes place with the text

foo bar. It becomes very simple to carry out rapid tests while dealing with Python and regex in the command line. You simply have to begin the python interpreter and carry out the importing of the re module as described in the above paragraphs. But, if you wish a Graphical User Interface to examine your Regular expression, you may carry out download written in the Python language by using the link given below: http:// svn.python.org/ view/* checkout*/ python/ trunk/ Tools/ scripts/ redemo.py? content-type = text% 2Fplain Various tools as well as desktop programs are available. They will be covered at some point in the coming chapters. For now, it is advisable to employ the interpreter for gaining fluency with them **Backslash in string literals** Res aren't a portion of the main Python language. Hence, there isn't a particular syntax for them and so they are dealt as any other string. The backslash character \ is employed to show metacharacters in Res. The backslash is also employed in strings for escaping particular characters. This implies that it has a distinct meaning in Python. Therefore, if we require to employ the backlash character, we have to escape it: \\. By doing so, the backlash is given the meaning of string literal. Though, for matching inside a regex, the backlashes should be escaped by efficiently writing 4 back slashes. To exemplify, a regex is written for matching \: > > > pattern = re.compile("\\\\") > > > pattern.match("\\ author") < *sre.SRE*Match at 0x104a88e68 > Since it can be seen, it is dull and not so easy for understanding in case the pattern is lengthy.
Python offers the string notation r. These raw string notation enable the backslashes to be considered as usual characters. Therefore, r"\ c" is not the backspace and represents the characters \ and c. This goes in a similar way for r"\n". The versions of python (2 and 3), deal with strings in a dissimilar manner. The 2.x version of python offers 2 strings types: eight-bit as well as Unicode; whereas the 3.x version of comprise of text and binary data. Text is mostly Unicode and the encoded Unicode is denoted as binary data. Strings have particular notation to show which type is employed.

Type	Explanation
String	String literals have been encoded automatically by employing the default encoding. The / is required to escape meaning characters.
Raw string	They are equivalent to literal strings with the exclusion of the backslashes which are considered as normal characters

Unicode Raw string	They are Unicode strings though they are considered as normal raw strings.

Building blocks for Python Regular Expressions: There are two distinct objects in python for handling regular expressions. One is known as RegexObject **.**It denotes a compiled regex and is also called Pattern Object. The second one denotes the matched pattern and is termed as MatchObject.

RegexObject

If we want to start matching patterns, we need the regular expression compilation. Python offers us an interface to achieve so. The outcome is a RegexObject. The RegexObject object has various methods for special operations on regex. The re module offers a shorthand for each task in order to enable us to ignore the compilation of it.
> > > pattern = re.compile(r'fo +') After compiling a regex, a recyclable pattern object is produced that offers all the operations which can be achieved like finding all substrings which match a specific regular expression, *etc*. Therefore, to exemplify, if we wish to know whether a string begins with <HTML>, we may employ the code given below: > > > pattern = re.compile(r' < HTML >') > > > pattern.match("< HTML >") < *sre.SRE*Match at 0x108076578
For the execution of tasks and matching patterns, there are two methods concerned with regex. You may either employ the compilation of a pattern that provides us a RegexObject or may employ the module operations. These two are compared in the example given below: > > > pattern = re.compile(r' < HTML >') > > > pattern.match("< HTML >") On the contrary, you may straightforwardly carry out the operation on the module by employing the line of code given below: > > > re.match(r' < HTML >', "< HTML >") The re module offers a wrapper for each task in the RegexObject. These may appear as shortcuts. Such wrappers produce the RegexObject , internally and after that call the apt method. You could be speculating if each time you are calling one of these wrappers it results in the compilation of the regex first. Well, the answer to this is no. The complied pattern is not cached, hence for the future calls this need not carry out the compilation again. Be careful of the memory requirements of your program. While you're employing module tasks, you are not controlling the cache, and therefore you may end up with a very high memory usage. You might always employ re.purge for clearing the cache though this is a compromise with

performance. By the compilation of patterns, you are enabled to have a better control of the memory usage since you may choose the right time of purging them. There are a few distinctions for both ways but. It becomes possible to restrict the area wherein the pattern might be searched with the RegexObject; to exemplify restrict the search of a pattern that lies between the characters at index 3 and 30.Additionally, you may fix flags in each call by employing tasks in the module. Though, be cautious; each time you alter the flag, a novice pattern would be assembled and cached.
Let us consider the most significant tasks that could be accomplished with a pattern object.

Searching
It is to be noted that the programming language python has two operations, match and search; most of the programming languages have one operation. Such a method makes an attempt to match the pattern that has been compiled at the starting of the string. A MatchObject is returned in case there is a suitable string. Hence, to exemplify, let's make an attempt to match if a string begins with <HTML>: < > > > pattern = re.compile(r' < HTML >') > > > pattern.match(" < HTML >< head >") < *sre.SRE*Match at 0x108076578 > In the above example, the compilation of the pattern takes place first of all and after that a match is discovered in the < HTML > < head > string. Now, we will see what the outcome is in case the string doesn't begin with < HTML>, as described in the code lines below: > > > pattern.match(" ⇢ < HTML >") None You can notice that there is no match. Match attempts to match at the starting of the string. Contrary to the pattern, the string begins with a whitespace. The distinction with search can be illustrated in the example given below: > > > pattern.search(" ⇢ < HTML >") < *sre.SRE*Match at 0x108076578 > As estimated, we get a match. The pos parameter tells where to begin searching, as described in the following code: > > > pattern = re.compile(r' < HTML >') > > > pattern.match(" ⇢ ⇢ ⇢ < HTML >") None <u>> > > pattern.match(" ⇢ ⇢⇢ < HTML >", 3) < *sre.SRE*Match at 0x1043bc890 ></u> In the underlined code, we may see the way the pattern has got a match even if there are three whitespaces in the string. This is so since we've set pos to three, therefore the match operation begins to search in that location.
The characters ^ and $ show the beginning and end of the string. You may not find them in the strings or write them, though they are always present and are legal characters for the regular expression engine. Note the distinct result when

we slice the string 3 positions, as given in the code below: > > > pattern.match(" ⇢ ⇢ ⇢ < HTML >"[3:]) < *sre.SRE*Match at 0x1043bca98
The slice provides us a novice string; hence, it consists of a ^ metacharacter in it. On the other hand, pos simply move the index to the beginning point for the search in the string.

Modifying the string: There are a number of operations in order to alter strings like an operation for the division of the string and others for swapping some portions of it.
split(string, maxsplit = 0) You may figure out the split operation in most of the programming languages. The main distinction is that the split in the re module has greater strength as a result of which you may use a regular expression. Therefore, in this case, the string is divided depending on the pattern matches.
Let’s try to understand this furthermore by employing an example, therefore let's fragment a string into lines: > > > re.split(r"\ n", "Ugly ⇢ is worse ⇢ than ⇢ beautiful.\ Implicit ⇢ is ⇢ worse ⇢ than ⇢ explicit.") [' Ugly ⇢ is ⇢ worse ⇢ than ⇢ beautiful.',Implicit ⇢ is ⇢ worse ⇢ than ⇢ explicit
In the above example, the match is \n; hence, the string is divided by employing it as the separator.
The ‘maxsplit’ parameter denotes the number of splits that can be achieved at maximum and returns the residual portion in the outcome: > > > pattern = re.compile(r"\ W") > > > pattern.split("Ugly is worse than beautiful", 2) ['Ugly, 'is', worse than beautiful]
As you can notice, just two words are divided and the other words are a portion of the outcome.

Match Object:
This object denotes the matched pattern; you may get one each time you carry out one of the following operations: Match
Search
finditer
Let's go through the most significant operations.

Group ([group1, …]) The group operation provides you the subdivisions of the match. When it's invoked without any argument or 0, it may return the entire match; whereas if 1 or more group identifiers are passed, the respective matches of the groups would be returned. Let's justify this by employing an example: > >

> pattern = re.compile(r"(\ w +) (\ w +)") > > > match = pattern.search("Hi ⇢ world") In this case, the entire string's pattern is matched and 2 groups are captured, Hi and world. After we are having the match, we may find the following real cases: · Without any arguments, the whole match is returned.
> > > match.group() 'Hi ⇢ world'
· With group one greater than zero, the respective group is returned.
> > > match.group(1) 'i'
> > > match.group(2) 'world'
· If there is no existence of the group, an IndexError will be given.
> > > match.group(4) … IndexError: no such group · With more than one argument, the respective groups are returned.
> > > match.group(0, 2) (' Hi ⇢ world', 'world') Here, we wish the entire pattern and the 2nd group, so we pass zero and two.
It is also possible to name the groups, we'll find it in detail in the coming chapter; there is a particular notation for it.
In case the pattern is having named groups, they might be accessed by employing either the names or the index: > > > pattern = re.compile(r"(? P < one>\ w +) (?P < two>\ w +)") In the above example, a pattern has been compiled for the capturing of 2 groups: the first one has been named as one and the second one has been named as two.
> > > match = pattern.search(" Hi ⇢ world") > > > match.group('one) 'Hi'
Hence, we may figure out a group via its name. It is to be mentioned here that by employing named groups we may still figure out the groups via their index, as described in the code below: > > > match.group(1) 'Hi'
We may employ both types: << match.group(0, 'first', 2) ('Hi ⇢ world', 'Hi', 'world') **Groups ([default])** The function of group is same as the preceding operation. Though a tuple is returned here that has all the co-groups in the match rather than offering you a single or a few of the groups. Let's understand this with an example explained in the preceding section.
>>>pattern=re.compile ("(\w+) (\w+)") >>> match=pattern.search ("Hi ⇢ world") >>>match.groups() ('Hi', 'world') In the preceding section, there were 2 groups 'Hi' and 'world' and that is what is delivered to us by groups in the form of group (1, lastGroup) When there are non-matching groups, there is a return of the default argument. In case the default argument has not been stated, then 'None' is employed. To exemplify: >>> pattern= re.compile ("(\w+) (\w+)?") Pattern.search("Hi ⇢ ") >>>match.groups ("munda") ('Hi', 'munda') >>> match.groups () In the above example, the pattern is attempting to

match 2 groups comprised of 1 or more alphanumeric characters. The 2nd one is optional; therefore we get only a single group with the string ‘Hi’. Once the match is obtained, **groups** are called with default that has been set to **munda.** Hence, the 2nd group is returned as **munda.**

Start ([group]) In some cases, it is helpful knowing the index where the matching of the pattern took place. Since with all the operations associated to groups, when the argument group is 0, then the operation functions with the entire string matched.
>>>pattern=re.complie(r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.start (1) 0
When there are groups having no match, then minus one is returned.
>>>math=pattern.search(“Hi ⇢ ”) >>>match..start (2) -1

Groupdict([default]) This method is employed in cases which employ named groups. A dictionary is returned by it with all the groups that are discovered.
>>>pattern=re.complie(r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.start (1) 2

Span([group])
It is an operation that offers you a tuple with the values from start and end. This operation is often employed in text editors to find and highlight a search. The code given below exemplifies this operation.
>>>pattern=re.compile (r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.span(1) (0, 2)

End([group])
This behavior of this operation is almost similar to start. The only difference being that this operation returns the end of the substring which the group matches.
>>>pattern=re.compile (r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.end(1) 2

Module Operations: Let us go through two helpful operations: **Escape()**
This operation escapes the literals which might occur while carrying out an operation.

Purge()

This operation is employed for purging the regex cache. It must be employed for freeing when the operations are employed through the module. It must be noticed that there exists a compromise with performance. When the cache is released, each pattern needs to be cached and compiled once again.

Summary

Now that you have gone through this chapter, you must have had an understanding of the various operations offered by Python and how the regular expression is handled by python. You must have been able to figure out the twists of Python while handling Res, the various string types, the Application Program Interface it provides via the MatchObject and RegexObject classes, each task which can be achieved with them in detail and some issues generally encountered by users. We hope that this chapter would have contributed well for you to have a better exposure to using regular expressions with python.

Assignment

Exercise 1

The output that is produced by programming languages that would be afterwards understood by an interpreter is termed as __________
Answer: Bytecode

Exercise 2

Choose the compilation flag that enables the writing of regex which are simpler to read and comprehend. To achieve this certain characters are handled in a specialized way.
a) re.LOCALE
b) re.VERBOSE
c) re.DEBUG
d) re.MULTILINE
The correct answer is (b)

Exercise 3

Choose the compilation flag that offers info about the pattern for compilation.
a) re.LOCALE
b) re.VERBOSE
c) re.DEBUG
d) re.MULTILINE
The correct answer is (c)

Exercise 4

Choose the compilation flag that is employed for altering the behavior of 2 metacharacters.
a) re.LOCALE
b) re.VERBOSE
c) re.DEBUG
d) re.MULTILINE
The correct answer is (d)

Exercise 5

Choose the compilation flag that is employed for matching the lower case as well as the upper case.
a) re.VERBOSE
b) re.DEBUG
c) re.MULTILINE
d) re.IGNORECASE
The correct answer is (d)

Exercise 6

Choose the metacharacter that is employed for matching the starting of the string and the starting of every novice line.
a) ^
b) $
The correct answer is (a)

Exercise 7

Choose the metacharacter that is employed to match the end of every line and every string. Categorically, it matches straight before the novel line character.
a) ^
b) $
The correct answer is (b)

Exercise 8

Choose the module operation that escapes the literals which might occur while carrying out an operation.
a) Escape()
b) Purge()
The correct answer is (a)

Exercise 9

Choose the module operation that must be employed for freeing when the

operations are employed through the module.
a) Escape()
b) Purge()
The correct answer is (b)

Exercise 10

What is the outcome of
>>>pattern=re.compile (r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.span(1)
Answer: (0, 2)

Exercise 11

What is the outcome of
>>>pattern=re.complie(r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.start (1)
Answer: 2

Exercise 12

What is the outcome of
>>>pattern=re.complie(r”(?P<one>\w+) (?P<two>\w+)?”)
>>>match=pattern.search(“Hi ⇢ ”) >>>match.start (1)
Answer: 0

Exercise 13

What is the outcome of
>>>pattern=re.compile (“(\w+) (\w+)”) >>> match=pattern.search (“Hi ⇢ world”) >>>match.groups()
Answer: (‘Hi’, ‘world’)

Exercise 14

What is the outcome of
> > > pattern = re.compile(r"\ W") > > > pattern.split("Ugly is worse than beautiful", 2) Answer: ['Ugly, 'is', worse than beautiful]

Exercise 15

Which one among the two denotes a compiled regex and is also called Pattern Object?

a) MatchObject

b) RegexObject

The correct answer is (b)

Exercise 16

Which one among the two denotes the matched pattern?

a) MatchObject

b) RegexObject

The correct answer is (a)

Chapter 3: Literals, Character Classes

Chapter Objective: The main objective of this chapter is to make the readers have an in depth understanding of literals, character classes. They are a very important part of regex coding and hence you need to have an in depth knowledge of it. We hope once you go through it, you will have a proper understanding of both these. In the end we have also explained dots, word boundaries and anchors so that the users have a better understanding of these important terms too.
As we are already familiar from chapter 1, literals are the easiest type of pattern matching in regexes. Whenever it discovers a literal, it will just succeed it.
Suppose are applying the regex*vixen* in order to find the phrase 'The fast brown vixen climbs over the lazy bitch', we will discover one match:
The ⇢ fast ⇢ brown ⇢ vixen ⇢ climbs ⇢ over ⇢ the ⇢ lazy ⇢ bitch *vixen*

Searching by employing a regex Though, we can also get various results rather than just a single one, if we use regex *am* to the phrase: I am what I am
I ⇢ am ⇢ what ⇢ I ⇢ am / **am** /
As we already are familiar that metacharacters can exist together with literals in one expression. As a result of this, we can search some expressions which fail to imply what they actually intended. To exemplify, if we are applying the expression/It is interior/, to find 'it is exterior' (it is exterior), we will discover that there are no parenthesis in the outcome. It occurs since the parenthesis have a specific sense and are metacharacters.
It ⇢ is exterior ⇢ (It ⇢ is ⇢ interior) /it ⇢ is ⇢ interior /

Wrongly unescaped metacharacters The metacharcters can be employed similar to literals. There are 3 ways to accomplish it:Escaping the metacharacters by preceding them by using a backslash. Employing the re.escape procedure to escape non-alphanumeric characters which might occur in the expression.The third method is to quote in regexes, the quoting with \E and \Q. For the languages that back it, it becomes very easy and all you have to do is enclose the portions that need to be quoted with \Q (it begins a quote) and \E (it finishes the

quote).
Unluckily, python does not back it for now.

Escaped metacharacters in regexes: In regexes, there exist 12 metacharacters which must be escaped in case they are to be employed with their literal meaning.
·'.'-Dot · **'\'-** Backlash ·**'$'-** Dollar sign ·'?'-Question mark · '(', ')'- opening as well as closing parenthesis · '[', ']'-opening as well as closing square brackets · '{'-opening curly bracket · '^'- Caret · '+'-plus sign · '|'-Pipeline symbol In a number of cases, the regex engines will do whatever they can to know whether they must have a literal meaning even though they are not escaped, to exemplify, the '{' will be considered as a metacharacter in case it is followed by a number to denote a repetition.

Character classes: We will employ a metacharacter, for the maiden time to understand the way the character classes are leveraged. The character sets or classes enable us to define a character which will map whether any of the well-defined characters on the set is there.
In order to describe a class, we must employ the '[' metacharacter and after than any accepted characters and in the last close it with a ']'. For example, we will define a regex which map the word "analyse" in British as well as American English.
analyze ⇢ and ⇢ analyse ⇢ are ⇢ valid *analy[zs]e*

Searching by employing a character class One can also employ the character's range. It is accomplished by using a '-' sign in between the two concerned characters; to exemplify, in order to map any lowercase alphabet, we can employ [a-z]. Similarly, to map a digit we might define the character set [0-9].
The ranges of the various character classes might be pooled to map a character against a number of ranges by simply putting a single range after the other without needing any particular separation. To exemplify, if we wish to map any uppercase or lowercase character, we may employ [0-9a-zZ-Z].
The table drawn below will aid you to understand this in a better way:

Element	Explanation
[	Maps the subsequent character set

0-9	Maps everything in between zero to nine
	Or
a-z	Maps everything between the alphabets a and z
	Or
A-Z	Maps everything between the alphabets A-B
]	Marks the termination of the character set.

One also has the option of the denial of ranges. The sense of a character set can be upturned by the placement of ‘^’ sign straight after ‘[’. In case we are having a character set class like [a-z] meaning any alphabet, the negated character class [^a-z] will map anything that is not a lowercase letter. Though, it is vital to understand that there should be a character which is not a lower case letter.

Predefined character classes: It would now have been clear to you that employing character classes makes it clear that a few of them are extremely helpful and possibly deserve a shortcut.
The good thing is that there is a variety of predefined character classes which employed again and again and the other programmers are already aware of them which make the expressions even more legible.
Such characters are not only helpful as famous shortcuts for specific character sets. Also they have different implications in various frameworks. The character class \w, that maps any character will map a character set reliant on the configured locale and the back of Unicode.
Below is a table that denotes the various character classes that are backed by python presently.

Element	Explanation for regular expression with default flags
.	Any character but s newline \n is mapped by it
\s	Any whitespace character is mapped by it

\S	Any non-whitespace character is mapped by it
\w	Any alphanumeric character is mapped by it
W	Any non-alphanumeric character is mapped by it

POSIX character classes in Python Various character classes denominations are offered by POSIX such as [:space:] for all whitespace characters which comprises of line breaks, *etc*. All of these classes follow the same representation which makes them easy to be identified. But they are not backed by python presently.
If you find one, you might implement the similar working by leveraging the functionalities of the character classes. The first element of the above table the dot '.' needs particular attention. It is possibly the eldest and the most widely employed metacharacter. Every character can be mapped by the dot but the newline character. The main cause why it does not map the newline is possible UNIX. The command line tools mostly operate line by line in case of UNIX and the regexes present at that time were applied distinctly to such lines. Hence, there were no novice newline characters to be mapped.
We will employ the dot by making regexes which map 3 characters having any value but the newline.

Element	Explanation
.	Maps any character
.	Maps any character followed by the preceding one

The dot metacharacter is a very strong metacharacter which can make problems when not employed properly. In a number of cases, it might be regarded as overkill (or only a sign of laziness while composing regexes).
To better explain what is needed to be mapped and to express more briefly to any hidden reader what a regex is envisioned to do, the usage of character classes is highly recommended. For example, while operating in Windows as well as UNIX file paths, in order to map any character apart from the backslash or slash, you might employ a negated character set: [^\/\]

Element	Explanation
[	Maps a character set
^	Not mapping this sign’s following characters
\/	Maps a / character
\	Maps a \ character
]	Finish of the set

This character is clearly revealing the fact that we wish to map anything except a UNIX or Windows file separator.

Repeating things: Being able to map various sets of characters is the maiden thing regexes can achieve which isn’t already achievable with the options accessible on strings. Though, if that is the sole extra capability of regular expressions, they might not be much of an advance. An extra ability is that you are able to specify that parts of the regular expression must be repetitive a fixed number of times.

The maiden metacharacter for repetition, that we will go through is ‘*’. It doesn’t map the literal character, rather it states that the preceding character can be mapped 0 or number times in place of precisely once.

To exemplify, ba*t will map ct (zero a characters), bat (one a), baaat (three a characters), and so on. The regular expression engine has different internal restrictions arising from the size of C’s int type which will stop it from mapping over two billion a characters; you possibly might not have sufficient memory to make a string that big, hence you must reach that limit.

Repetitions like ‘*’ are greedy; while repeating a regular expression, the

mapping engine will make an attempt to echo it as many times it can. In case later parts of the pattern don’t map, the mapping engine would then back up and attempt it again with some repetitions.
A side-by-side example will make this more obvious. Let’s consider the expression b[cde]*c. This maps the letter ‘b’, 0 or more letters from the class [cde], and finally finishes with a ‘c’. After this, assume mapping this regular expression against the string bcbde.
Alternation:
In this section we will learn how to match against a set of regexes. This is done by employing the pipe sign ‘|’. Let’s begin by saying that we wish to map the word “yes” or the word “no”. By employing it becomes as easy as: *yes|no*
The table will give you a furthermore understanding of what we mean:

Element	Explanation
	Maps either the given sets of characters
yes	Maps the characters y, e and s
|	Or
No	Maps the characters n,o

On the contrary, if we require to accept greater than 2 values, we may go on adding them in the manner shown below: *yes| no| maybe*

Element	Explanation
Yes	The word “yes”
|	Or
No	The word “no”
|	Or

Maybe	The word “maybe”

When employing huge regexes, we would probably require to wrap alternation inside ‘()’to show that just that portion has been alternated and not the entire expression. To exemplify, when me commit the mistake of not employing ‘()’as in the expression given below: *License: yes|no*

Element	Explanation
	Maps either of the given sets of characters
License	The characters L,i,c,e,n,s,e,:, ⤏
Yes	y,e and s
|	Or
No	The characters n and o

We might consider that we accept License: yes or License: no, but in reality we accept either License: yes or no since the alternation has been put to the entire regex rather than merely on the yes|no portion. A right approach for this should be: Alternation options /Driving ⤏ License: ⤏ (yes|no)/
Demarcated alternation **Regexes employing alternation Dot**

The dot maps one character, not being thoughtful what that character implies. The only exception are line break characters .This exception exists mostly due to important reasons. The maiden tools that employed regexes were line-dependent. They might recite a file line by line, and apply the regex distinctly to every line. The result is that by employing these tools, the string might never comprise line breaks, hence the dot might never map them.
Current tools and languages might apply regexes to very huge strings or even whole files. Excluding for JavaScript and VBScript, all regular expression flavors deliberated here offer an option to make the dot map all characters, plus line breaks In Perl, the style wherein the dot also maps line breaks is known as "single-line mode". It is a bit unlucky, since it is simple to blend this term with

"multi-line mode Other flavors and regular expression libraries have accepted Perl's terms. When employing the regular expression classes of the .NET framework, this approach is triggered by stipulating RegexOptions .Singleline,.
JavaScript as well as VBScript are not consisting of a choice to make the dot map line break characters. In such languages, you might employ a character class like [\s\S] to map any character. This character maps a character which is a whitespace character, or a character which is not a whitespace character. As all characters might be whitespace or non-whitespace, this character maps any character.
Whereas back for the dot is common among regular expression flavors, there are noteworthy dissimilarities in how they handle line break characters. All flavors consider the noviceline \n just like a line break. UNIX info files dismisses lines with one noviceline.

Anchor:
Anchors are a distinct breed. They do not map any character at all. Rather, they map a location beforehand, afterward, or amid characters. They might be employed to "anchor" the regular engine map at a particular position. The ^ symbol maps the location beforehand the maiden character in the string. Using ^b to bcd maps b. ^b does not map bcd at all, since the c cannot be mapped straight after the beginning of the string, mapped by In the same way, $ maps straight after the end character in the string. d$ matches d in bcd, whereas b$ will not map at all.
A regular expression that comprises merely of an anchor might only find 0-length maps. This could be helpful, however can also make problems.

Word boundary
As we have already learnt that the \b is a metcharacter that is employed for matching a location termed as **word boundary**. Such a match is 0-length.
The following are the three different locations in order to be considered as word boundaries ·In case the first character is a word character, the location qualifying as a word boundary is before it ·In case the last character is a word character, the location qualifying as a word boundary is after it.
·The location of the word boundary is in between 2 string characters wherein the first is a word character whereas the second one is not so.
In easy terms: \b enables you to perform a "entire words only" search employing a regex in the case of \bword\b. A "word character" might be defined as a

character that can be employed to create words. "Non-word characters" can be considered as the antonym of word character Precisely which characters will be word characters rests on the regular expression flavor you're operating on. In a number of flavors, characters which are mapped by the short-hand character class \w are the characters which are considered as word characters by word boundaries.

In a number of flavors, excluding the ones conversed below, just have a single metacharacter which maps both beforehand a word and afterward a word. This is so since any location between characters could never be both at the beginning and at the last of a word. Employing only a single operator makes things simpler for you.

As digits are supposed to be word characters, \b4\b can be employed to map a 4 which is not portion of a higher number. This regular expression does not map 44 sheets of a4. Hence, saying "\b maps beforehand and afterward an alphanumeric sequence" is more precise than telling "beforehands and afterwards a word".

\B is the annulled version of \b. \B maps at each location where \b does not. Efficiently, \B maps at any location between 2 word characters and at any location between 2 non-word characters.

Summary

Now that you have gone through this chapter, we hope that you have an in depth understanding of literals, character classes. They are a very important part of regex coding and hence you need to have an in depth knowledge of it. We hope that now you will have a proper understanding of both these.

Chapter 4: Quantifiers and Boundary Matchers

Chapter Objective:
The main objective behind this chapter is to make the readers familiar with quantifiers and boundary matchers. This chapter explains in detail the various concepts related to boundary quantifiers including the main types of quantifiers like possessive quantifiers, greedy as well as reluctant quantifiers. The chapter also explains boundary matchers in detail. We hope that once you go through this chapter, you will be able to have a good understanding of them.

Quantifiers:
Till now, we went through different ways of defining a character in a number of ways. Now will learn about quantifiers-the methods to define the way a character, character set or metacharacter might be recurred.
To exemplify, when we define that a \d could be recurring a number of times, we might simply make a form validator for the 'number of times' field (don't forget that \d maps any decimal). However, let us begin from the starting, the 3 main quantifiers: '?', '+' and '*'.
The table below will give you a furthermore understanding of them:

Name	**Element**	**Quantification of the preceding character**
Asterisk sign	*	0 or more times
Curly brackets	{a,b}	Between a or b times
Plus sign	+	1 or more times
Question mark sign	?	Optional (zero or one times)

In the table above, we can have a glimpse on the 3 main quantifiers, all of them with a particular use. For example,'?' (Question mark) can be employed to map the word bat and its plural form bats: *bats?*

Element	**Explanation**
Bat	Matches the characters b, a, t and s
S?	Maps the character ‘s’ on choice

In the example given above, the ‘?’ is simply applied to ‘s’ and not to the entire word. The quantifiers are mostly applicable to the preceding token. Another exciting instance of the application of the ‘?’ quantifier would be to map a telephone number which could be in the format 666-666-666,666, 666 666 666 or 666666666.
We are now are aware about the character sets can be leveraged to accept various characters, however it possible that a quantifier is applicable characters, their sets and also can be applicable to groups. We can create a regex such as the one we created above in order to validate the phone numbers.
/ \d [-\s]? \d+[- \s]? \d+ /
The table given below gives a better explanation of this regex:

Type	**Element**	**Explanation**
Already defined set of characters	\d	Any random decimal
Quantifier	+	- which is repeated once or more
Set of characters	[-\s]	A ‘-’ or whitespace character
Quantifier	?	- Which can or cannot appear
Already defined set of characters	\d	Any random decimal
Quantifier	+	- which is repeated once or more
Any defined character set	\d	Any random decimal
Quantifier	+	- which is repeated once or more

At the starting, one or more type of quantifiers employing the ‘{}’ brackets had been shown. By employing this syntax, we might define that the preceding character should appear precisely 3 times adding it with {3}, which is expression \w {9} shows precisely nine alphanumeric digits.
We might also define a particular range of repetitions by offering a minimum and maximum times of repetitions which is between 3 and 9 times could be defined with the syntax {3,9}. Either the highest and lowest value could be omitted defaulting to zero and infinite respectively. To do a repetition of up to 4 times, we might employ {,4}, we could also do a repetition for a minimum of four times with {4,}.
Rather than employing {,1}, you might employ ‘?’. The same goes for {0,} for the ‘*’ and {1,} for ‘+’.
Other programmers will want you to accomplish so. When you don’t follow this practice, everyone going through your expressions would lose some attempting to find out what types of fancy material you tried to achieve.
These 4 various combinations are shown in the table given below:

Syntax	**Explanation**
{a}	The preceding character is repeated precisely ‘a’ times
{a,}	The preceding character is repeated at least ‘a’ times
{,a}	The preceding character is repeated ‘a’ times at the maximum
{a,b}	The preceding character is repeated between a and b times (both inclusive)

Earlier, we made a regex to validate phone numbers which can be in the format 666-666-666, 666 666 666, or 666666666. We defined a regex to validate it by employing the metacharacter ‘+’: /**d+ [-\s]? \d+[-\s]?\d** /. It will need the digits (**d**) to be reiterated 1 or more times.
We will now further learn in the regex by defining that the extreme left cluster of digits can comprise up to 4 characters whereas the remaining digit clusters must contain 4 digits.
Exactly 4 repetitions

/\d{1,4}[-\s]? \d {4} [-\s]
Between one and four repetitions ***Using Quantifiers***

Greedy as well as reluctant quantifiers Still we have not defined what will be mapping when we apply a quantifier like /".+"/ to a text like: English "sweet", Spanish " dulce". We might expect it maps "sweet" and "dulce", but it would actually map "sweet", Spanish "dulce".
This is known as greedy and is either of the two probable behaviors of the quantifiers in Python language. The first one is termed greedy behavior. It will attempt to map as much as probable to accomplish the biggest possible map. The non-greedy behavior is denoted by the addition of an extra '? ' to the quantifier. To exemplify, *?, ??. A quantifier denoted as reluctant will act just like the antonym of the greedy type. They will attempt to have largest map possible.

Possessive Quantifiers: This is another behavior of the quantifiers named as the possessive quantifiers. It is backed by the Java and .NET flavors of the regexes at present.
They are denoted by an additional '+' sign to the quantifier; to exemplify, *+, ? +. Possessive quantifiers shall not be covered furthermore in this book. We might have a better understanding how this quantifier operates by going through the same regex (with the exclusion of leaving the quantifier) to the same type of text that has two highly distinguishing outcomes.
".+?"
English ⇢ "sweet", ⇢ Spanish ⇢ "dulce"
".+?"

Greedy as well as reluctant quantifiers Boundary Matchers:
Up to here we have only attempted to figure out regexes within a text. Sometimes it is needed to map an entire line, we might also require to map at the starting of a line or even at last. This could be accomplished because of the boundary matchers. They are a variety of identifiers which will correspond to a specific position in the interior of the input. The table given below gives an overview of the boundary matchers present in python:

Matcher	**Explanation**
$	Maps at the starting of a line

\b	Maps at word boundary
\B	Maps the antonym of \b. Everything is not a word boundary.
\A	Maps the starting of the input
\Z	Maps the last of the input
^	Maps the starting of the line

They will act distinctly in distinct contexts. To exemplify, the word boundaries (\b) will rely straightly on the configured locale since various languages can have various word boundaries and the starting and end of the line boundaries shall behave distinctly depending on particular flags which we will go through in the coming chapter.
Let us have a furthermore understanding of the boundary matchers by composing a regex which will map lines which begin with "Name". When you have a look at the preceding table, you will be figuring out the presence of a metacharacter ^ which expresses the starting of a line. By employing it, we may compose the following expression.
^*Name:*

Element	**Description**
^	Maps the starting of the line
N	Maps the followed by N
A	Maps the followed by a
M	Maps the followed by m
E	Maps the followed by e
:	Maps the followed by :

When we wish to take one step forward and keep on employing the '^' and the '$' symbol in combination to map the end of the line, we must also keep in mind that from this moment we will map against the entire line and not merely attempting to find a pattern in the line.
Learning from the preceding example, let us consider we wish to assure that after the name there exist just alphabetic characters till the finish of the line. This will be done by us by mapping the whole line upto the last by setting a character set with the permitted characters and enabling their recurrence any number of times till the finish of the line.
^*Name: [\sa-zA-Z]+$*

Element	Explanation
^	Maps the starting of the line
N	Maps the followed by character N
a	Maps the followed by character a
M	Maps the followed by m
E	Maps the followed by e
:	Maps the followed by ':'
[\sa-zA-Z]	Now maps the followed by uppercase or lowercase character,
+	The character is 1 or more number of times
$	Till the finish of the line

Another remarkable boundary matcher is the word boundary \b. It is used for

mapping any character which is not a word character and hence and possible boundary. This is extremely helpful when we wish to work with isolated words and we do not bother to create sets of characters with each single character that might distinguish our words. We might for example, assure that the word ‘sweet’ is present in a text by employing the following regexes.
sweet

Element	Explanation
\b	Maps the word boundary
S	Maps the followed by character s
W	Maps the followed by character w
E	Maps the followed by e
E	Maps the followed by e
T	Maps the followed by t
\b	Now maps another followed by word boundary

Why is the preceding expression better than *sweet*? The answer for this question is that the expression will map an isolated word rather than a word containing “sweet” which implies that *sweet*would simply map sweet, sweeted or otsweet; whereas *sweet* will simply map sweet.

Summary

Now that you have gone through this chapter, you must be familiar with quantifiers and boundary matchers. You must have been able to learn in detail the various concepts related to boundary quantifiers including the main types of quantifiers like possessive quantifiers, greedy as well as reluctant quantifiers. You must have also had good understanding of boundary matchers in detail. We hope that now you will be able to have a good understanding of them.

Assignment

Exercise 1

Choose the one that is the methods to define the way a character, character set or metacharacter might be recurred.

a) Quantifier

b) Boundary Matcher

Answer: (a)

Exercise 2

What does {a,} imply?

a) The preceding character is repeated at least ‘a’ times b) The preceding character is repeated precisely ‘a’ times c) The preceding character is repeated between a and b times (both inclusive) d) The preceding character is repeated ‘a’ times at the maximum Answer: (a)

Exercise 3

What does {,a} imply?

a) The preceding character is repeated at least ‘a’ times b) The preceding character is repeated precisely ‘a’ times c) The preceding character is repeated between a and b times (both inclusive) d) The preceding character is repeated ‘a’ times at the maximum Answer: (c)

Exercise 4

What does {a,b} imply?

a) The preceding character is repeated at least ‘a’ times b) The preceding character is repeated precisely ‘a’ times c) The preceding character is repeated between a and b times (both inclusive) d) The preceding character is repeated ‘a’ times at the maximum Answer: (c)

Exercise 5

What does What does {,a} imply?

a) The preceding character is repeated at least ‘a’ times b) The preceding character is repeated precisely ‘a’ times c) The preceding character is repeated between a and b times (both inclusive) d) The preceding character is repeated ‘a’ times at the maximum Answer: (b)

Exercise 6

What is used in case it is needed to map an entire line, we might also require to map at the starting of a line or even at last?
Answer: Boundary matchers

Chapter 5: Special characters, module contents and Regular Expression Objects

Chapter Objective:

The main objective of this chapter is to make the readers well aware of special characters, module contents and regular expression objects. The chapter gives an overview of almost all of them and tries to give the readers an idea of all of them as they are important part of using regular expressions in python. The chapter also explains various module contents and regex objects in detail. We hope that after going through this chapter, you will have a good understanding of all these important concepts.

The special characters are:

(Dot .)
In the standard mode, this maps any character but a newline. In the event of the DOTALL flag been stated, this maps any character plus a novice line.
(dollar $: Maps the end of the string or just before the novice line at the last of the string, as well as in MULTILINE mode also maps afore a novice line too maps both ‘too’ and ‘toobar’, whereas the regex too$ maps only ‘too’. More importantly, searching for too.$ in 'too1\ntoo2\n' maps ‘too2’ normally, but ‘too1’ in MULTILINE mode; finding for a single $ in 'too\n' would find 2(empty) matches: first just afore the novice line, and one at the last of the string.
(Asterisk): Makes the resulting regex to map zero or more repetitions of the earlier regex, as many repetitions as are conceivable. bc* will map ‘b’, ‘bc’, or ‘b’ followed by any number of ‘c’s.
'+': Makes the resulting regex to map one or more repetitions of the earlier regex. bc+ will map ‘b’ followed by any non-zero count of ‘c’s; it will not map just ‘b’.
'**?**': Makes the resulting regex to map zero or one repetitions of the following regex. bc? Will map either ‘b’ or ‘bc’.

*?, +?, ??
The '*', '+', and '?' qualifiers are all greedy; they map as much text as probable. Sometimes this behavior isn’t wanted; when the regex <.*> is mapped against '<H1>title</H1>', it will map the whole string, and not simply '<H1>'. Totaling '?' after the qualifier makes it do the map in non-greedy or minimal fashion; as less characters as it can be mapped. Employing.*? In the last expression will map only '<H1>'.
{m}: States that precisely m copies of the preceding regex must be mapped ; lesser maps cause the whole regex not to map. To exemplify, a{7} will map precisely seven 'a' characters, but not six.
{m,n}: Makes the resulting regex to map from m to n repetitions of the previous regexes , trying to map as many repetitions as is l. For example, a{3,5} will match from 3 to 5 'a' characters. Omitting m specifies a lower bound of zero, and omitting n specifies an infinite upper bound. As an example, a{4,}b will match aaaab or a thousand 'a' characters followed by a b, but not aaab. The comma may not be omitted or the modifier would be confused with the previously described form.
{m,n}?: Makes the resulting regex to map from m to n repetitions of the previous regexes, trying to map as less repetitions as it make . It is the non-greedy type of the preceding qualifier. Toe exemplify, on the six -character string 'bbbbbb', b{3,5} will map five 'a' characters, whereas b{3,5}? will only map three characters.
'\': Either results in the escape of the special characters (allowing you to map characters such as '*', '?', and so on), or hints a particular sequence.
In case you’re not employing a raw string to show the pattern, don’t forget that Python also employs the backslash as an escape series in string literals; when the escape series isn’t acknowledged by Python’s parser, the ‘\’and following character are encompassed in the resultant string. But, in case Python will identify the resultant sequence, the ‘\’ must be repeated two times. It is complex and difficult to understand, therefore it’s extremely suggested that you employ raw strings for all but the easiest expressions.
[]: It is employed to show a set of characters. You may list characters on an individual basis such as, [ank] will map 'a', ‘n’, or 'k'. It is possible to denote the ranges of characters by offering two characters and partitioning them via a '-', to exemplify [b-z] will map any lowercase ASCII letter from b to z, [0-6][0-9] will map all the two-digits numbers from 00 to 69, and [0-9A-Fa-f] will map all the hexadecimal digits. When there is escaping of - (e.g. [a\-z]) or when it’s

positioned as the maiden or the ending character (e.g. [b-]), it will map a literal '-'. When inside sets, special characters do not have their specific meaning. To exemplify, [(+*)] will map any of the literal characters '(', '+', '*', or ')'.
Character classes like \w or \S are also received inside a set, even though the characters they are mapping relies on if LOCALE or UNICODE mode is in effect. Complementing the set maps the characters, which are not contained in between the range. If the first character of the set is '^', all the characters that are not in the set will be matched. To exemplify, [^6] will map any character but '6', and [^^] will map any character but '^'. ^ has no particular meaning in case it’s not the maiden character in the set.
To map a literal ']' in a set, a ‘’it with a ‘\’, or place it at the starting of the set. To exemplify, [()[\]{}] and []()[{}] will map a parenthesis.
'|': N|M, where N and M can be random regexes, makes a regex which will map either N or M. A random number of regexes can be partitioned by the '|' in this manner. This can be employed in the interior of the groups also. Since the target string is scanned, regexes partitioned by '|' are attempted from left to right. In case a pattern totally maps, that branch is accepted. This implies that once N matches, M will not be verified furthermore, even though it shall result in a lengthier overall map. To be exact, the '|' operator is not a greedy operator. To map a literal '|', employ \|, or enfold it in the interior of a character class, such as [|].
(...): Maps whatever regex is in the interior ‘()’, and shows the beginning and end of a group; the insides of a group can be recovered after a map has been carried out, and can be mapped afterwards in the string with the \number special series, shown below. To map the literals '(' or ')', employ \(or \), or encircle them in the interior of a character class: [(] [)].
(?...): It denotes extension representation. The maiden character after the '?' controls what the sense and future syntax of the construct is. Most of the times extensions do not make a novice group. Below are the presently supported extensions.
(?iLmsux)
The group maps the empty string; the alphabets set the following flags: re.U (Unicodereiant), re.I (ignore case), re.S (dot maps all), re.M (multi-line), , re.L (locale reliant), re.X (verbose), for the whole regex.
(?:...)
It is a non-capturing type of normal parentheses. Maps whichever regex is in the interior of the parentheses, however the substring mapped by the group cannot

be recovered after executing a match.
(?P<name>...): They are just like to regular parentheses, though the substring mapped by the group is accessible through the representative group name. Group names should be legal Python identifiers, and every group name should be defined only one time within a regex. A representative group is also a numbered group, simply in case the group was not named.
Named groups can be referenced in three contexts. If the pattern is (?P<quote>['"]).*?(?P=quote) (i.e. matching a string quoted with either single or double quotes).

Module Contents
The module defines a number of things such as functions. A few of the functions are easier are versions of the complete featured methods for compiled regexes. Maximum of the non-trivial applications mostly employ the compiled type.
re.compile(pattern, flags=0)
Compile a regex pattern into a regex object, that can be employed for mapping by employing its match() and search()types , explained below.
The expression’s actions can be changed by giving the values of flags. Values could be the variables given below, combined by employing ‘|’ operator).
The sequence
code= re.compile(pattern)
outcome= prog.match(string)
is same as
outcome= re.match(pattern, string)
but employing re.compile() and saving the resultant regex object for using again more well-organized when the expression would be employed various times in one program.
It is to be noted that the compiled types of the most newest patterns forwarded to, re.search()are cached, hence codes that employ just a few regexes at a time don’t require to be concerned about compiling regexes.
re.A
re.ASCII
Make the special sequences do ASCII-only mapping rather than complete Unicode mapping. It only makes sense for Unicode patterns, and is not given much consideration for byte patterns.
It must be noted that in order to have backward compatibility, the re.U flag still are there and re.UNICODE and its implanted counterpart (?u)), though these are

recurring in the third version of Python 3 as maps are Unicode by standard for strings (and Unicode mapping isn’t permitted for bytes).
re.DEBUG
Shows debug info about the expression that is assembled.
re.I
re.IGNORECASE
Does case-insensitive mapping; expressions such as [A-Z] will map lowercase letters, also. It is not altered by the present locale and runs for Unicode characters as anticipated.
re.L
re.LOCALE
Make the special sequences such as \b, \s reliant on the present locale. The usage of this flag is not appreciated as the locale process is very untrustworthy, and it only takes care of a single “culture” at any time in any case; you must employ Unicode mapping rather, this is the default in the third version of Python for Unicode (str) types.
re.M
re.MULTILINE
'^' maps at the starting of the string and at the starting of every line (following every novice line instantly) and '$' maps at the last of the string and at the last of every line (preceding every novice line instantly). By standard, '^' maps only at the starting of the string, and '$' just at last of the string and instantly before the noviceline (if any) at last of the string.
re.S
re.DOTALL
Enables the '.' map any character at all, which contains with a noviceline; without this flag. This special character will map everything but a noviceline.
re.X
re.VERBOSE
This flag enables you to compose regexes that appear better. Whitespace in the pattern is unobserved, but when in a character class and in case a line does not have a '#' in a character class or all characters from the extreme left like '#' through the finish of the line are unnoticed.
That implies that the 2 below regexes objects that map a decimal number are have same workingl: c = re.compile(r"""\d + # the integral part

```
\.            # the decimal point
\d *  # some fractional digits""", re.X)
```

be= re.compile(r"\d+\.\d*")
re.search(pattern, string, flags=0)
Inspect via string finding for a position where the regex pattern produces a map, and revert back respective match object. Reverts back None in case there is not any position in the string maps the pattern; it must be taken into consideration that it is distinct from finding a 0-length map at any instant in the string.
re.match(pattern, string, flags=0)
In case 0 or more characters at the starting of string map the regex pattern, revert back a respective match object. Revert back none in case the string does not map the pattern; it is to be considered that this is distinct from a 0-length match.
Again, it is to be considered that even in MULTILINE mode, re.match() will simply map at the starting of the string and not at the starting of every line.
In case you wish to find a map anywhere in string, employ search() re.split(pattern, string, maxsplit=0, flags=0)
Partitions string by the occurrence of pattern. In case capturing parentheses are employed in pattern, the text of every group in the pattern is also reverted back as part of the consequential list. In case maxsplit is not zero, at the maximum maxsplit splitstakes place, and the remaining part of the string is reverted back as the final part of the list.

```
>>>
>>> re.split('\W+', 'Bords, bords, bords.')
['Bords', 'bords', 'bords', '']
>>> re.split('(\W+)', 'Bords, bords, bords.')
['Bords', ', ', 'bords', ', ', 'bords', '.', '']
>>> re.split('\W+', 'Bords, bords, bords.', 1)
['Bords', 'bords, bords.']
>>> re.split('[a-f]+', '0a4B9', flags=re.IGNORECASE)
['0', '4', '9']
```

In case there is an occurrence of capturing groups in the separator and it maps at the beginning of the string, the outcome will begin with an empty string. This is similar for the end of the string: >>>

```
>>> re.split('(\W+)', '...bords, bords...')
['', '...', 'bords', ', ', 'bords', '...', '']
```

This way, separator elements are always established at the similar relative indices within the result list.
It is to be considered that split would never split a string on an empty pattern match. To exemplify: >>>

```
>>> re.split('x*', 'boo')
['boo']
>>> re.split("(?m)^$", "boo\n\nbar\n")
['boo\n\nbar\n']
```
re.findall(pattern, string, flags=0)

Reverts back all non-overlapping maps of pattern in string like a strings list. The string has been scanned left-to-right, and maps are reverted back in the order discovered. In case 1 or more groups are there in the pattern, revert back a list of groups; this would be a list of tuples in case the pattern comprises of more than a single group. Empty matches are comprised in the outcome until they touch the starting of another match.
re.finditer(pattern, string, flags=0)
Reverts back an iterator giving map objects over all non-overlapping maps for the regex pattern in string. The string has been scanned left-to-right, and maps are reverted in the order discovered. Empty maps are comprised in the outcome until they touch the starting of another map.
re.sub(pattern, repl, string, count=0, flags=0)
Revert back the string gotten by changing the leftmost non-overlapping happenings of pattern in string by the replacement repl. In case the pattern isn’t there, string is reverted back unaltered. repl could be either a string or a function; in the event of it being a string, any backslash escapes in it are refined. It means, \n is transformed to a single noviceline character, \r is transformed to a carriage return, and *etc.* Unknown escapes like \j are left aloof. Backreferences, like \7, are changed with the substring mapped by group 7 in the pattern. To exemplify: In case, repl is a function, it is known as for each non-overlapping happening of pattern. The function takes a one match object argument, and reverts back the changed string. To exemplify: >>>
```
>>> def dashrepl(matchobj):
...             if matchobj.group(0) == '-': return ' '
...             else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Maked means as well as Spam', flags=re.IGNORECASE) 'Maked means as well as Spam'
```
The pattern might be a string or an regex object.
The optional argument number is the highest count of pattern occurrences to be

changed; count should be a non - integer. If nil or 0, all occurrences would be changed. Empty maps for the pattern are changed merely in case not neighboring to a preceding map, therefore sub('x*', '-', 'cde') reverts back '-c-d-e-'.
In addition to the character escapes and backreferences as explained in the preceding sections, \g<name> will employ the substring mapped by the group titled name, as defined by the (?P<name>...) syntax. \g<number> employs the respective group number; \g<2> and hence same as \2, but isn't vague in a replacement like \g<3>0. \30 would be understood as a reference to group 30, not a reference to group 3 trailed by the literal character zero. The backreference \g<0> substitutes in the whole substring mapped by the regex.
re.subn(pattern, repl, string, count=0, flags=0)
Do the similar operation like nsub(), but revert back a tuple (new_string, number_of_subs_made).
re.escape(string)
Revert back string with all non-alphanumerics backslashed; this is helpful in case you wish to map an random literal string which might have regex metacharacters in it.
re.purge()
Eliminate the regex cache.
exception re.error
Exception occurs in case a string passed to 1 of the functions here is not a suitable regex (to exemplify, it could comprise unmapped parentheses) or when some other error happens while compilation or mapping. There is no error in any case a string comprises no map for a pattern.

Regular Expression Objects
Compiled regex objects back the below methods and attributes:
regex.search(string[, pos[, endpos]])
Scan via string looking for a position in which this regex creates a map, and reverts back a respective match object. Revert back 'None' in case no location in the string maps the pattern; it is to be considered that this is distinct from discovering a 0-length map at a particular point in the string.
The elective second parameter pos offers an index in the string in which the search is to begin; it defaults to zero. It is not totally equal to slicing the string; '^' matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.
The elective parameter endpos restricts how large the string would be searched;

it would be like the string is endpos characters in length, hence just the characters from pos to endpos - 1 would be found for a map. In case endpos is lower than than pos, there will not be any match found ;in other case , in case rx is a compiled regexp object, rx.search(string, 0, 60) is same as rx.search(string[:60], 0).

```
>>>
>>> pattern = re.compile("d")
>>> pattern.search("cow")
<_sre.SRE_Match object at ...>
>>> pattern.search("cow", 1)
```

regex.match(string[, pos[, endpos]])

If 0 or more characters at the starting of string match this regex, revert back a respective match object. Revert back ‘None’ in case the string does not map the pattern; it is to be considered that this is distinct from a 0-length match.

The elective pos and endpos parameters have the similar sense as for the search() process.

```
>>>
>>> pattern = re.compile("o")
>>> pattern.match("cow")
>>> pattern.match("cow", 1)
<_sre.SRE_Match object at ...>
```

If you wish to find a map anywhere in string, employ search() as an alternative.

regex.split(string, maxsplit=0)

Similar to the split() function, employing the compiled pattern.

regex.findall(string[, pos[, endpos]])

Just like the findall() function, employing the compiled pattern, it also accepts elective pos and endpos parameters that limit the search region like for match().

regex.finditer(string[, pos[, endpos]])

Similar to the finditer() function, employing the compiled pattern, but also accepts optional pos and endpos parameters that restrict the search region such as for match().

regex.sub(repl, string, count=0)

Similar to the sub() function, by employing the compiled pattern.

regex.subn(repl, string, count=0)

Similar to the subn() function, by employing the compiled pattern.

regex.groups: The count of capturing clusters in the pattern.

regex.groupindex

A dictionary matching any figurative group names characterized by (?P<id>) to group counts. The dictionary is null in case no symbolic groups were employed in the pattern.
regex.pattern
The pattern string wherein the regular expression object was compiled.

Summary

Now that you have gone through this chapter, you must be well aware of special characters, module contents and regular expression objects. You must have had an overview of almost all of them and have an idea of all of them as they are important part of using regular expressions in python. You also might be having various module contents and regex objects in detail. We believe that now you will have a good understanding of all these important concepts.

Assignment

Exercise 1

Which special character maps any character but a newline?
Answer: Dot

Exercise 2

Which special character maps the beginning of the string, and in MULTILINE mode also maps directly after every newline?
Answer: Caret

Exercise 3

Which special character maps the end of the string or just before the novice line at the last of the string?
Answer: Dollar

Exercise 4

Which special character makes the resulting regex to map zero or more repetitions of the earlier regex, as many repetitions as are conceivable?
Answer: Asterisk

Exercise 5

Which special character makes the resulting regex to map zero or one repetitions of the following regex?
Answer: Question mark

Exercise 6

Which between the two states that precisely m copies of the preceding regex must be mapped?
a) {m},
b) {m,n}
Answer: (a)

Exercise 7

Which among the following make the special sequences do ASCII-only mapping rather than complete Unicode mapping?

a) re.ASCII

b) re.DEBUG

Answer: re.ASCII

Chapter 6: Grouping

Chapter Objective:
The main objective is to make the readers have a further more understanding of using regular expressions with python**.** The main point covered in this topic is grouping. First of all the chapter explains expressions with grouping so that the reader becomes well familiar with their use. After this the chapter explains back references and how they can be used more efficiently. Next the chapter deals with different types of groups like named groups and atomic groups and explains how they can be employed to make the use of regular expressions in an easier manner. The last section deals with specialized cases with groups which include yes/no pattern and overlapping patterns.
Grouping is an emphatic technique that enables you to carry out operations like:
Formulating sub expressions that can be practiced for quantifiers. To exemplify, the reusing a subexpression instead of employing a single character.
Rather than changing the entire expression, we may describe precisely what needs to be changed.
Gathering data from the matched pattern.
Employing the extracted data again in regular expression, which is possibly the most helpful property. An example would be the detection of recurrent words.

Expressions with Python:
Grouping is achieved via two metacharacters and (). The easiest instance of employing parenthesis would be the creation of a subexpression. To exemplify, consider you have a having a product list, the identity for every product comprises of 2 or 3 series of digits that are followed by a '-' and an alphanumeric character, 1-a3-b: >>>re.match(r "((\d-(\w{2,3}",ur"1-a3-b")
<_sre.SRE_Match at 0x10f690798>
As can be noticed from the previous example, the '()' show that the regular expression engine which the parenthesis include inside them must be considered as a single unit.
Let us consider one more example. Here we have to match in case there is 1 or more bc followed by d: >>>re.search(r "(bc)+d", ur "abbcd")
<_sre.SRE_Match at 0x10f69a09
>>>re.search (r "(bc)+d", ur "abbc")

There is another characteristic of grouping which is termed as capturing. You may employ groups in various operations since they capture matched pattern.
To exemplify, consider that there is a product list. The identities of the product list (ID) comprise of numbers which denote the state to which the product belongs, a '-' as a separator and 1more alphanumeric as the identity in the database. A request is made for the extraction of the state codes.

```
>>>pattern= re.compile (r "((\d+)-\w+")
>>>it= pattern.finditer(r "1-a\n30-baer\n44-afcr") >>>match=it.next ()
>>>match.group (1)
'1'
>>>match=it.next ()
>>>match.group (1)
'30'
>>>match=it.next ()
>>>match.group (1)
'44'
```

In the previous example, we have formulated a pattern in order to match the identities, though we only capture groups comprising of the state digits. Don't forget while working with the group method, the entire match is returned by the index zero as well as the group begins at index one. Capturing offers a large variety of responsibilities owing to which they can be employed with a variety of operations which would be studied in the coming sections.

Back references:
You already might be having an idea of what back references are. We will have a furthermore understanding of them in this section. As has been described, one of the most emphatic functionalities of grouping is that grouping enables us to use the group that has been captured inside the regular expression or other operations. This is actually what the back references offer. Possibly the most important example for bringing some clarity is the regular expression to search duplicated words as described in the example given below:

```
>>>pattern=re.compile (r "(\w+) \1") >>>match=pattern.search(r"hi hi world")
>>>match.groups ()
('hi',)
```

In this case, a group compiled of 1 or more than 1 alphanumeric character is matched. After this a whitespace is attempted to be matched by the pattern and ultimately we are having the \1 back reference. It has been underlined and

highlighted above which implies that it must match precisely the similar thing it matched as the 1st group.
They can be employed with the first ninety nine groups. Of course, with a rise in the group number, you may discover the job of reading and maintaining the regular expression in a more complicated manner. It can be decreased by employing named groups. We will have an understanding of them in the coming sections and before that we need to learn a few things more about back references. Hence let us consider a furthermore operations in back references wherein the back references are employed. Rememorize the example described in the previous section wherein there was a product list. Let us attempt to alter the order of the identity, so we are having the identity in the database, a -, and a country code.

```
>>>pattern=re.compile (r "(\w+)")
>>>it= pattern.finditer(r "\2-\1", "1-a\n30-baer\n44-afcr") ('a-1\nbaer-30\nafcr-44')
```

This is not so complicated! Is it? It is to be mentioned here that the identity in the database is also being captured so that it can be employed afterwards. The highlighted and underlined code implies that whatever has been matched with the 2nd group, a '-', and the 1st group is to be replaced.
Employing numbers might not be easy to follow and also sustain. Therefore let us go through what is provided by python via the re-module to aid this.

Named groups:
It is hoped that you understand the example from the previous chapter: > > > pattern = re.compile(r"(\ w +) (\ w +)") > > > match = pattern.search("Hi ⇢ world") > > > match.group(1)
'Hi'
> > > match.group(2)
'world'
Now we already have an understanding of the way of accessing groups by employing indexes for the extraction of data and also employing it as back references. Employing numbers to mention groups could be difficult and pretty confusing and the worst case scenario is that it does not assist to provide meaning to the group. Hence the groups have been named.
Consider a regular expression wherein you are having a number of back references (say 20) and you discover that the 3rd one is not valid. Therefore you delete it from the regular expression. This implies that you have to alter the

index for each back reference beginning from that one onwards. For resolving this issue, Guido Van Rossum in the late ninety nineties discovered named groups for Python 1.5. This was also provided to Perl for the purpose cross-breeding.

In the present scenario, it can be discovered in any every flavor. Basically, it enables us to offer names to the groups enabling them to be known by their names each time a group is used. For employing it every time we have to employ the syntax (?P<name>pattern), wherein the P arrives from the specialized python extensions.

We will now see how this actually operates with a previously mentioned example in the given code piece.

```
>>>pattern= re.compile (r “(?P <one>\w+)”) (?P<two>\w+)”)
>>>match=re.search (“Hi world”)
>>>match.group (“one”)
‘Hi’
>>>match.group (“two”)
‘world’
```

Hence the maintenance of backreferences is easy and usage is really simple as is obvious in the example described below: >>>pattern= re.compile (r “(?P<Country>\d+)-(?P<identity>\w+)”) >>>pattern.sub (r “\g<identity>-\g<country>”, “1-a\n30-baer\n44-afcr”) ‘a-1\nbaer-30\nafcr-44’

As can be noticed from the above example, for the sake of referencing a group by the name in the sub-operation, we must employ \g<name>.

We may also employ named groups in the pattern. This is described in the example given below: >>>pattern=re.compile (r “?P<word>\w+”) (?P=word)”)

```
>>>match=pattern.search (r “hi hi world” )
>>>match.groups ()
(‘hi’,)
```

This is easier and definitely clearer as compared to using numbers.

In all these examples, we have employed the 3 different methods of mentioning named groups.

Used	Syntax
It is employed inside a pattern	(?P=name)
It is employed in MatchObject’s tasks	Match.group (‘name’)

It is employed in the sub operation’s repl string	\g<name>

Atomic groups:

They can be considered as special instances of non-capturing groups. Normally, they are employed for improving the performance. It deactivates backtracking, hence with them you may get rid of cases where trying all possibilities in the pattern does not seem appropriate. It is not so easy to be understood. The re module does not back atomic groups. Hence, for seeing an example, we will employ the regular expression module.

Consider, we need to look for an identity comprising of 1 or more than 1 alphanumeric character trailed by a ‘-’ and a digit.

```
>>>info = “aabbaaccdddddddaaaaaa”
>>>regex.match (“(\w+)-\d”, info)
```

We will see what is happening here one by one:

1. The 1st is a matched by the regular expression engine.
2. After this each character is matched till the string finishes.
3. It is unable to find ‘-’, hence it fails.
4. Hence the regular expression engine implements backtracking and attempts the matching again with the next a.
5. Perform the same steps again.

This is attempted with each character. When you carefully analyze what we are attempting, you will be forced to think that there is no point of attempting it again when we have not been successful in the first attempt. This is where the atomic groups find their application.

```
>>>regex.match  (“(?>\w+)-\d”,info)
```

In this case,?> has been added which shows an atomic group. Hence, when the regular expression engine is unsuccessful in matching, it will not keep on attempting with every left character in the info.

Specialized cases with groups:

Python offers with groups that can aid us to change the regex or match a pattern if only the preceding group is there. For example: if statement.

Yes or No pattern:

It is a very helpful case of groups. It attempts to match a pattern if there exists a preceding one for it. On the contrary, no match is attempted if there is no preceding group is discovered. To be precise, it is similar to an ‘if-else

statement’. Its syntax is given below: (?(id/name) yes-pattern|no-pattern) This implies when the group with this identity is matched, then the yes pattern needs to match. On the contrary, if there is no match, the y-pattern must be matched.
Let us figure out how it functions. There is a product list. It is to be noted that the identity (ID) can be created by two distinct methods.
·The code of the country which is 2 digit, a ‘-’, 3 or 4 alphanumeric letters, a ‘-’ and the code of the area which is again a 2 digit number. To exemplify: 44-def1-01.
·3 or 4 alphanumeric letters. To exemplify, def1.
Hence in case there is a country code, the country area must be matched.
```
>>>pattern=re.compile (r” (\d\d-)? (\w {3,4}) (? (1) (-\d\d))”) >>>pattern.match (“44-def1-01”)
<sre.SRE_Match at 0x10f68b7a1>
>>>pattern.search (“erte”)
<_sre.SRE_Match at 0x10f68b829>
```
You might notice it from the preceding example; a match is found when the country and area code are given. Also, there is no match in case there is no area code, while the country code is still there.
```
>>>pattern.search (“44-defl”)
None
```
Now we will add one more limitation to the preceding example, if there is no country code, then there should be a name when the string concludes.
```
>>>pattern =re.compile (r “(\d\d-)? (\w {3, 4})-(? (1) (\d\d) |[a-z]{3,4}) $” ) >>>pattern.match (“44-def1-22”)
<_sre.SRE_Match at 0x10f6ee740>
```
As anticipated, when there is a country code as well as an area code, the outcome is that a match has been found.
```
>>>pattern.match (“44-def1”)
None
```
In the previous example, there is a country code but not any area code, hence there does not exist any match.
```
>>>pattern.match (“def1-fghi”)
<_sre.SRE_Match at 0x10f6ee888>
```
In this case, there is no country or area, but fghi name is there. So a match if found.
It is to taken into consideration that the ‘no-pattern’ is selective. Hence we have not used it in our example.

Overlapping groups

Overlapping can be very confusing at times and it seems to confuse a number of people. Hence, we will try to make it clearer with an easy example.

>>>re.findall (r ‘(b|c) +’, ‘bcbdb’) [‘b’, ‘b’]

The characters bcd are matched, but only b comprises of the captured group. It is so as even if our regular expression has grouped all the characters, it will stay with the b that is in the end. It must not be forgotten that since it explains how it functions. Halt for some time and imagine it, the regular expression is requested that it has to capture the groups comprising of b or c, but only for a single character among all and that is the main point. Hence, what is the way of capturing groups that have been comprised of several ‘b’ and ‘c’ in any order?

The catch here is the expression given below:

>>>re.findall (r ‘(?:b|c)+)’, ‘bccbdb’) [‘bccb’, ‘b’]

In this, we are actually making a request to the regular expression engine to capture each of the group that comprises of the subexpression (b|c) and not group merely a single character.

Also, it is to be noted that in case we want to have each group which comprises of b or c with findall, we might compose this easy expression, >>>re.findall (r ‘(b|c)’, ‘bcbdb’)

[‘b’, ‘c’, ‘b’, ‘b’]

Here a request is made to the regular expression engine to capture a group which comprises of b and c. Since we are employing findall, we have each pattern matched; hence we are having 4 groups.

Note: It is more advisable to keep regex as easy it can be. Hence, you must start with the easiest expression and after that create more complicated expressions in a stepwise manner.

Summary

The third chapter may seem an easy one but you must make sure that you are thorough with all the basics explained in this chapter. They will definitely aid in dealing with regular expressions in a better manner. Now that you have gone through this chapter, you must have been able to have a good understanding of expressions with regular expressions. You must have been familiar with groups like named groups, atomic groups and have a better exposure to backreferences. Lastly, the chapter introduces special cases with regular expressions and so you must have been able to know how to use yes/no pattern and as well as how to deal with overlapping in a better way.

Assignment

Exercise 1

What will be the outcome for >>>pattern=re.compile (r “(\w+) \1”) >>>match=pattern.search(r“hi hi world”) >>>match.groups () Answer: (‘hi’,)

Exercise 2

What is the result of the outcome of >>>pattern=re.compile (r “(\w+)”) >>>it= pattern.finditer(r “\2-\1”, “1-a\n30-baer\n44-afcr”) Answer: (‘a-1\nbaer-30\nafcr-44’)

Exercise 3

What is the outcome for > > > pattern = re.compile(r"(\ w +) (\ w +)") > > > match = pattern.search("Hi ⇢ world") > > > match.group(1) Answer: ‘Hi’

Exercise 4

What is the outcome of >>>pattern=re.compile (r “?P<word>\w+”) (?P=word)”) >>>match=pattern.search (r “hi hi world”) >>>match.groups () Answer: (‘hi’,)

Exercise 5

What is the outcome of >>>pattern =re.compile (r “(\d\d-)? (\w {3, 4})-(? (1) (\d\d) |[a-z]{3,4}) $”) Answer: None

Exercise 6

What is the outcome of >>>re.findall (r ‘(b|c) +’, ‘bcbdb’) Answer: [‘b’, ‘b’]

Exercise 7

Which one among the following will be the outcome of >>>re.findall (r ‘(b|c)’, ‘bcbdb’) Answer: [‘b’, ‘c’, ‘b’, ‘b’]

Exercise 8

What will be the outcome for >>>re.findall (r ‘(?:b|c)+)’, ‘bccbdb’) Answer: [‘bccb’, ‘b’]

Chapter 7: Look Around

Chapter Objective: The main objective of this chapter is centered at making the readers familiar with the concept of look around. The chapter introduces the idea of zero-with assertions and the way it can be helpful to discover the precise thing in a text not interfering in the content of the outcome. The chapter also leverages the 4 kinds of look around processes: positive look ahead, positive look ahead, negative look ahead as well as negative look behind. Once you have gone through this chapter, you must be able to have an idea of these are to be employed and manipulated in the regular expressions used in python.

Till now, you must have been familiar with mechanisms to match characters when they are being cast-off. It is not possible to match a character that has already been matched. In order to match it it has to be discarded.
A contradiction to this a variety of metacharacters : the zero-width assertions. They show positions instead of the real content. For example, the (^) sign denotes the starting of a line or the ($) symbol stands for the finish of the line. They simply make sure whether the location of the input is right without really consuming any character.
A comparatively stronger type of the zero-width assertion is look around. It is a process in which enables the matching of some particular look behind which means the preceding one or look ahead which means the hidden one to the present position. They perform assertion efficiently not consuming characters; they simply return an affirmative or negative outcome of the match.
The process of look around is possibly the most unknown and yet the strongest mechanism in regex. This process enables us to make strong regexes which cannot be created otherwise. The reason for this can be complication that it might offer or simply due to technical boundaries of regexes without look around.
This chapter mainly deals with leveraging the look around process by employing Python regexes. We will learn the way these function and the few boundaries that they are having. Types of look ahead and look behind: Positive Look Behind: This process is denoted as an expression preceded by a '?', '<' sign, '=' clustered inside a parenthesis block. To exemplify, the (?<=regex) shall map whether the passed regular expression does map with the preceding input.

Negative Look Ahead: This process is denoted as an expression that is preceded by a ‘?’ and an ‘!’ inside the ‘()’ block. To exemplify, (?!regex) maps whether the passed regular expression does not match map against the upcoming input.
Positive Look Ahead: This method is denoted as an expression preceded by a ‘?’ and ‘=’ symbol. To exemplify, (?=regex) shall map whether the passed regular expression does map against the upcoming input.
Negative Look Behind: This process is denoted as an expression preceded by ‘?’, ‘<’ and ‘!’. To exemplify, (?<!regex) will map whether the passed regular expression does not map against the preceding input.

Look Ahead
The maiden look around process that we will go through is the look ahead process. It attempts to match the ahead the subexpression that has been conceded as an argument. As we are already familiar from the preceding section, it id denoted as an expression that is preceded by a ‘?’ and ‘=’ symbol wrapped inside a ‘()’.
Let’s have a furthermore understanding of this by making a comparison between two regexes. Consider the example: >>>pattern=re.compile (r’vixen’) >>>result=pattern.search (“The fast brown vixen climbs over the lazy bitch”.) >>>print result.start (), result.end () 17 20
We simply searched ‘vixen’ in the input string and it was searched in between the index ‘17’ and ‘20’. No we will go through the how the look ahead process works.
>>>pattern=re.compile (r ‘(?=vixen)’) >>>result=pattern.search (“The fast brown vixen climbs over the lazy bitch”) >>>print result.start (), result.end () 17 17
Since the (?=vixen) is applied, the outcome is simply a position at the 17 index. It is so since look around is not absorbing characters and hence can be employed for filtering where the expression must be matched.
To have a furthermore understanding of this we will again attempt and match a word trailed by a ‘,’ by employing the regex /r’\+(?=,)’/ as well as the text. They were three: Henry, Cursor; and Barem: >>>pattern= re.compile (r ‘\w+(?=,)’) >>>pattern.findall (“They were three: Henry, Cursor, and Barem.”) [‘Henry’, ‘Cursor’]
We made a regex which accepts any repetition of characters trailed by a ‘,’ character, which is not being employed as a portion of the outcome. Hence, simply Henry and Cursor appeared as an outcome since Barem did not have a “,”

following the name.
In order to have any idea of how distinct it was as compared to the regexes we went through till now in this chapter. We will have a comparison of the outcomes by applying /\w+,/ to the text.
>>>pattern=re.complie (r ‘\w+,’) >>>pattern.findall (“They were three: Henry, Cursor and Barem.”) [‘Henry’, ‘Cursor’]
With the preceding regexes, we asked the regexes to take any repetition of characters followed by a ‘,’ character. Hence the character and ‘,’ character would be returned.
It is to be noted here that the look ahead process is another process which can be leveraged with all the strength of the regexes. Hence, we might employ all the constructions that we went through till now as the alteration.
>>> pattern= re.compile (r ‘\w+ (?=.)’) >>>pattern.findall (“They were three: Henry, Cursor and Barem”) In the previous example, we employed alternation, but we could have other easier techniques in order to accept any recurrence of the characters trailed by a ‘,’ or ‘.’ character which is not going to be employed as a part of the outcome.
Negative Look Ahead: This process offers a similar nature as compared to the look ahead though with a noteworthy difference: the outcome would be legal when subexpression is not matching.
It is helpful in case we wish to show what must not happen. To exemplify, to search any name Jahn that is not Jahn Dmith, we might do as described below: >>>pattern=re.compile (r ‘Jahn (?!\sBmith)’) >>>result= pattern.finditer (“I shall rather out go out with Jahn McLume than with Jahn Bmith or Jahn Con Movi”) >>> for i in result:… print i.start (), i.end () …

```
27 31
63 67
```

In the previous example, we searched for Jahn by consuming all the 5 characters and after that for a whitespace character trailed by word Bmith. If there is a match, it will comprise of only the beginning and the final position of Jahn. Here they are 27-31 for Jahn McLume as well as 63-67 for Jahn Con Mavi.
Now that we have an understanding of the basics of look around, let us have a furthermore understanding of how to employ in them substitutions and groups.
Substitutions and Look around: For substitutions, the zero width nature of the look around operation is extremely helpful. As a result of them, we are able to do transformations that shall be otherwise be highly complicated for both reading and writing.

One main example of this would be changing numbers comprising of simply numeric characters like 12345, into a ‘,’ separated number which means: 1, 234, 5.
For writing this regex, we have to have a plan to follow. We intend to cluster the numbers in blocks of 4 which will then be replaced by the same group plus a ‘,’ character.
We might easily begin with a raw approach with the below underlined regex.
>>>pattern= re.compile (r ‘\d{1,4}’) >>>pattern.findall (“The number is 4567890145”) [‘4567’, ‘8901’, ‘45’]
We have not been successful in this try. We are efficiently grouping in blocks of four characters, but they should be considered from the right to the left. We require another approach. Let us attempt to search 1, 2, 3 or 4 digits which need to be followed any block number of 4 digits till we discover something which is not a digit.
It will affect the number as follows: While attempting to discover 1, 2 or 3 digits, the regex will attempt to take only 1 and it will be number one. After this, it will attempt to catch blocks of precisely four numbers, for example, 4567890145 till it discovers a non-digit. This is the finish of the input. If we show a regex what we now explained in simple English, we shall get something given below: *\ d {1,4} (?= (*d {4}) + (?!\d))/

Look Behind:
Look Behind can be easily defined as the antonym of look ahead. It attempts to match behind the subexpression conceded as an argument. Also, it has a 0-width nature and hence will not be a portion of the outcome.
We might employ it in an example comparable to the one we employed in negative look ahead to search simply the surname of somebody named Jahn McLume. In order to achieve this, we might compose a look behind the one given below: >>>pattern=re.compile (r’ (?=<=Jahn\s) McLume’) >>>result=pattern.finditer (“I shall rather out go out with Jahn McLume than with Jahn Bmith or Jahn Con Movi”) >>>for i in result: … print i.start (), i.end () …
32 28
With the previous look behind, we demanded the regular expression engine to match only those expressions that Jahn as well as white space to then consume McLume as an outcome.
There is a distinction in the basic module of python in the way look ahead and

look behind are put into implementation. Owing to number technical causes, the look behind process is just able to match fixed-width patterns. They do not have variable length matchers like backreferences are not permitted either. Alternation is enabled but only when alternatives are having the similar length. Also, these restrictions are not there in abovementioned regular expression module.
There is an exception since look behind needs a fixed-width pattern. A similar result will be gotten by us when we attempt the application of quantifiers or other altering length constructions.
Since, we are familiar with the methods of matching ahead or behind the while not using characters and the various limitations we might encounter, we might attempt to compose another example that embraces a few of the processes which have learnt to solve a real-world issue.

Negative Look Behind: This process offers extremely similar nature of the look behind process. But there is an effective outcome only when the passed subexpression is not mapping. It is worth noticing that negative look behind shares the various features of the look behind mechanism as well as its restrictions. The negative look behind process is merely able to match fixed-width patterns. They are having the similar cause and consequences that we went through in the preceding section.
We might have a practical implementation of this by attempting to match any person who has been surnamed as Boe and is not named Jahn with a regex like: /(?<!Jahn\s) Boe/. If we employ it in Python's console, we will get the result given below: >>>pattern=re.compile (r'(?<!Jahn\s) Doe') >>>results=pattern.finditer ("Jahn Boe, Calvin Boe, Hobbey Boe") >>> for result in results: … print result.start (), result.end () …
17 20
29 32

Look around and groups: There is an important application of look around constructions and it is the inside groups. Generally when groups are employed, a very precise outcome needs to be mapped and returned to the group. Since we do not pollute the groups with the data that is not needed, in other potent groups, we might leverage look around as a promising answer.
Suppose, we wish to have a comma partitioned value, the maiden portion of the value is a name whereas the 2nd is a value, it will be having a format like: INFO 2014-09-17 12:15: 44, 488

As we have already that we may easily acquire these 2 values as: /\w+\s[\d-]+\s[\d:,]+ \s(.*\sfailed)/
Though, we simply wish to match in case there is an authentication failure.
We might achieve this by summing a negative look behind. It will appear as follows: /\w+\s[\d-]+\s[\d:,]+ \s(.*\? <! Authentication failed\s) failed) /
When this is applied to python’s console, the below input will be achieved:

```
>>>pattern=re.compile (r ‘\w+\s[\d-]+\s[\d:,]+ \s(.*(?<!authentication failed )’)
>>>pattern.findall (“INFO 2014-09-17 12:15:44, 448 authentication failed) []
>>>pattern.findall (“INFO 2014-09-17 12:15:44,448” something else was unsuccessful) [‘Something else was unsuccessful’]
```

Summary

Now that you have gone through this chapter, you must have been able to have an idea about zero-with assertions and the way it can be helpful to discover the precise thing in a text not interfering in the content of the outcome.
We also went through how to leverage the 4 kinds of look around processes: positive look ahead, positive look ahead, negative look ahead as well as negative look behind.
After going through this chapter, the travel through the fundamental and advanced techniques around regexes is expected to have been concluded for the readers.

Assignment

Exercise 1

What will be the outcome of >>>pattern= re.compile (r '\w+(?=,)') >>>pattern.findall ("They were three: Henry, Cursor, and Barem.") Answer: ['Henry', 'Cursor']

Exercise 2

What will be the outcome of >>>pattern= re.compile (r '\d{1,4}') >>>pattern.findall ("The number is 4567890145") ['4567', '8901', '45']

Exercise 3

What will be outcome of >>>pattern=re.compile (r '\w+\s[\d-]+\s[\d:,]+ \s(.*(?<!authentication failed)') >>>pattern.findall ("INFO 2014-09-17 12:15:44, 448 authentication failed) Answer: [ ]

Exercise 4

What will be the outcome of >>>pattern.findall ("INFO 2014-09-17 12:15:44,448 " something else was unsuccesful) ['Something else was unsuccessful']

Exercise 5

Which one among the following is denoted as an expression preceded by a '?', '<' sign, '=' clustered inside a parenthesis block?

a) Positive Look Behind b) Negative Look Ahead c) Positive Look Ahead d) Negative Look Behind The correct answer is (a)

Exercise 6

Which one among the following is denoted as an expression that is preceded by a '?' and an '!' inside the '()' block?

a) Positive Look Behind b) Negative Look Ahead c) Positive Look Ahead d) Negative Look Behind The correct answer is (b)

Exercise 7

Which one among the following is denoted as an expression preceded by a ‘?’ and ‘=’ symbol?

a) Positive Look Behind b) Negative Look Ahead c) Positive Look Ahead d) Negative Look Behind The correct answer is (c)

Exercise 8

Which one among the following is denoted as an expression preceded by ‘?’, ‘<’ and ‘!’ ?

a) Positive Look Behind b) Negative Look Ahead c) Positive Look Ahead d) Negative Look Behind The correct answer is (d)

Chapter 8: Measuring performance of regexes

Chapter Objective:
The main objective of this chapter is to give the readers an understanding of performance. Firstly the chapter deals with benchmarking a regular expression in python and for the sake of benchmarking our regular expression, the time is calculated that a regular expression takes to implement. It is vital that they are to be tested with distinct inputs, since with minor inputs; almost all regular expressions are very quick. The chapter also provides a furthermore understanding of the DEBUG flag and debugging in regex buddy tool. Lastly, it makes the readers familiar with how backtracking enables heading back and getting the distinct paths of the regex and how it is employed.

Till now, we were mainly centered at knowing the way a feature is leveraged to achieve an outcome without bothering too much about how quick the process is going to be. Our main aims were accuracy as well as readability.
The fifth chapter mainly deals with an entirely different issue that is performance. Though we will discover that most of the times the increase in performance will ruin the readability. When we are changing something to make it quicker, we are actually making it simpler for the machine to comprehend and hence we are possibly compromising on human readability.
How to benchmark regexes with python:
For the sake of benchmarking our regular expression, we shall calculate the time a regular expression takes to implement. It is vital that they are to be tested with distinct inputs, since with minor inputs; almost all regular expressions are very quick. Though with lengthier ones, it might be totally different. We will have a furthermore understanding of this in the backtrack section.
Now, we will make a small function to aid us with this task: >>>from time import clock as now

```
>>>def test (f, args, *kargs): Start=now ()
f (args, *kargs)
print “The function %s stayed”:
%f “ %(f.___name___, now () - start)
```

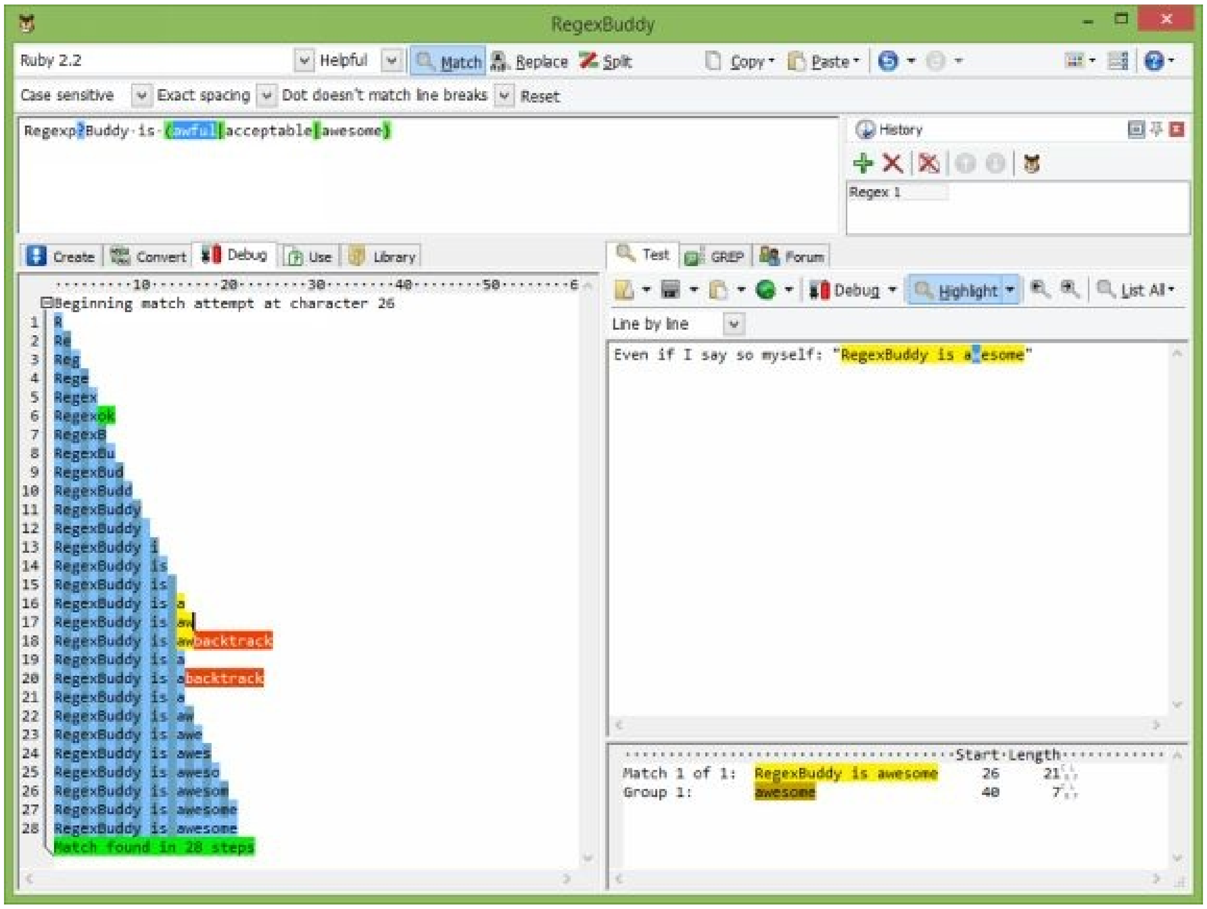
Hence, we may examine a regular expression by employing the code given below: >>>def change(text):
pat=re.compile (‘spa (in|niard)’) pat.search (text)
>>>test (change, “spain”)
The function change stayed:
0.000009
There is an inherent profiler in python that can be employed to calculate the time and count the calls.
>>>import cProfile
>>>
cProfile.run (“change (‘spaniard’)”)
Let us go through another helpful method which is going to aid you when you want to examine what is happening under the hook of your regular expression.
We will have a furthermore understanding of this by using the flag DEBUG which we are already familiar with: The DEBUG flag provides us the info about the way a pattern is to be assembled. To exemplify: >>>re.compile((‘\w+\d+)+-\d\d’,re.DEBUG) max_repeat 1 4294967296

```
subpattern  1
max_repeat 1
4294967296
 in
            category
category_word
max_repeat 1
4294967296
in
  category
  category category_digit
            literal 45
in
category category_digit
in
category category_digit
```

It can be noticed here that there are 3 max_repeat conditions from 1 to 4294967296, 2 of them are in another max_repeat.
They can be considered as nested loops. But this is not so advisable and it might result in a catastrophic backtracking. We will go through it in the coming

sections.

The RegexBuddy Tool:
As compared to all other tools that are there for improved productivity while composing regexes, the RegexBuddy Tool is an excellent option. It has been developed by Just Great Software Co. Limited. The main person behind this being Jan Goyvaerts. Regexbuddy enables us to employ visual interface for creating, examining and debudgging regexes. The debug characteristic is almost exclusive and offers a good process to access how the regex engine is operating behind the scenes. The picture given below shows the Regexbuddy debugging the regex's implementation.



```
Regexp?Buddy is (awful|acceptable|awesome)
```

```
·········10·········20·········30·········40·········50·········6
Beginning match attempt at character 26
 1  R
 2  Re
 3  Reg
 4  Rege
 5  Regex
 6  Regexok
 7  RegexB
 8  RegexBu
 9  RegexBud
10  RegexBudd
11  RegexBuddy
12  RegexBuddy
13  RegexBuddy i
14  RegexBuddy is
15  RegexBuddy is
16  RegexBuddy is a
17  RegexBuddy is aw
18  RegexBuddy is awbacktrack
19  RegexBuddy is a
20  RegexBuddy is abacktrack
21  RegexBuddy is a
22  RegexBuddy is aw
23  RegexBuddy is awe
24  RegexBuddy is awes
25  RegexBuddy is aweso
26  RegexBuddy is awesom
27  RegexBuddy is awesome
28  RegexBuddy is awesome
    Match found in 28 steps
```

```
Even if I say so myself: "RegexBuddy is awesome"
```

```
···································Start·Length···········
Match 1 of 1:   RegexBuddy is awesome      26      21
Group 1:        awesome                    40       7
```

RegexBuddy debugging a regex
It certainly has other characteristics such as a library of generally employed regexes as well as a code generator for various programming atmospheres. There are certain shortcomings that is comes with. Its license is copyrighted and the only build obtainable is for Microsoft windows. Though, the implementation on

Linux by employing the wine emulator is there.

Going through the Python regular expression engine: The re-module employs a backtracking regex engine. The most important characteristics of the algorithm are listed below: · It backs lazy quantifiers like *?, ?? and +?

· It maps the maiden co-incidence though there are lengthier ones in the string. This implies that order is critical.

· The algo tracks only a single transition in one step that implies that engine checks a single character at a single incidence of time.-

· It backs '()' as well as backreferencing.

· Backtracking is the capability of recalling the last productive position so that it might head back and retry if needed.

· In the poorest scenario, the complexity is exponential.

Backtracking

As has been described previously backtracking enables heading back and getting the distinct paths of the regex. This is performed by recalling the last productive position. It applies to quantifiers as well as alternation.

An established issue with backtracking is known as catastrophic backtracking that can offer you various issues ranging from a slow regular expression to a crash with a stack overflow. The issue comes into picture when in case the regular expression is not successful in matching the strings. We will benchmark a regular expression with a method we have encountered formerly so that we can have a better understanding of the issue.

Let us attempt an easy regular expression:

```
>>>def catastrophic (n):
Print "Testing with %d characters" %n
Pat=re.compile ('b' +'c')
Text= "%s" % ('b' *n)
            Pat.search (text)
```

Since there is not the presence of any 'c' in the end, the text that is being attempted to match is never going to be successful.

We shall now test it various inputs.

```
>>>for n in range (30, 40):
Test (catastrophic, n)
Testing with 30 characters
The function lasted: 0.130457
```

```
Testing with 31 characters
The function lasted: 0.245125
……
The function lasted: 14.828221
Testing with 38 characters
The function lasted: 29.830929
Testing with 39 charcters
The function lasted: 61.110949
```

The conduct of this regular expression appears as if it were quadratic. The matter here is that (b+) turns greedy, therefore it attempts to fetch as many b characters as possible. Then it fails to match the c which means that it backtracks to the second and continues consuming b characters till it fails to map c. After that, it attempts the whole process again beginning with a second b character.

Let us go through another example. An exponential behavior for this case: Def catstrophic (m):

```
Print "Testing with %d characters" %m
Pat= re.compile ('(x)+(b)+c')
Text='x'*m
Text+='b'*m
Pat.search (text)
Form in range (12,18)
Test (catastrophic, m)
Testing with 12 characters:
The function catastrophic lasted: 1.035161
Testing with 13 characters:
The function catastrophic lasted: 4.084713
Testing with 14 characters:
The function catastrophic lasted: 16.31916
Testing with 15 characters:
The function catastrophic lasted: 65.855181
Testing with 16 characters:
The function catastrophic lasted: 276.941307
```

As can be seen, the behavior is exponential that might result in exponential outcomes.

.

Recommendations for Optimization:

In the coming sections, there will be a variety of recommendations which could be applied to improvise regexes. The most effective tool would be common sense, ofcourse and it will be required even when recommendations are to be followed. It needs to be understood that once the recommendation is to be applied and when not. To exemplify, you cannot apply the ‘don’t be greedy’ recommendation for every instance.

Reuse Compiled Patterns: We have previously explained that in order to employ a regex, we need to transform it from its string representation to a form called RegexObject.
A little time is consumed by such kind of compilation. If we are employing the remaining module rather than employing the compile function to ignore making the RegexObject, we have to understand that the compiling is implemented anyway and various compiled RegexObject are cached automatically. Though while we are performing the compilation,, that cache will not support us. Each compile execution will take an approximately insignificant amount of time for a single execution, but it is certainly relevant if a number of executions are carried out.
Let us have a look at distinction between reusing and not reusing the compiled patterns in the example given below: >>>def dontreuse():

```
Pattern=re.compile (r ‘\bfoo\b’) Pattern.match (“foo bar”)
>>>def
Callninehundredninetyninetimes ():
For_in
range (999):
don’treuse()
>>>
test(callninehundredninetynine times)
The function callninehundredninetyninetimes lasted: 0.001964
>>>pattern=
Re.compile (r ‘\bfoo\b’) >>>def reuse():
Pattern.match (“foo bar”)
>>>def
call ninehundredninetyninetimes():
for_in
range (999):
reuse()
>>>
```

Test(callninehundredninetyninetimes)
The function
Callninehundredninetyninetimes lasted: 0.000632
>>>

Extraction of general portions in alternation In regexes, alternation is mostly performance risk. While employing them in Python, we must gather any common part on the external of the alternation.
For example, if we are having /(Hi ⇢ World|Hi ⇢ Continent|Hi ⇢ Country), we might conveniently extract Hi ⇢ with this expression: Hi ⇢ (world|continent|country)/. This shall allow our engine to simply check Hi ⇢ once and this shall not return to check again for every probability.
\>>>pattern=re.compile (r'/(Hi\sWorld│Hi\sContinent│Hi\sCountry') >>>def **nonoptimized** (): pattern.match ("Hi\sCountry")
>>> def callninehundredninetyninetimes (): for_in
range (999):
nonoptimized ()
>>>
test (callninehundredninetyninetimes)
The function callninehundredninetyninetimes) The function callninetyhundredninetyninetimes lasted: 0.000644
>>> pattern=re.compile (r '/Hi\s(World│Continent│Country)') >>> def optimized ():
Pattern.match ("Hi\sCountry")
>>> def ninehundredninetynine ():
For_in
Range (999):
Optimized ()
>>>
Test (callninehundredninetyninetimes)
The function callninetyninehundredninetytninetimes : 0.000543
>>>

Shortcut to alternation:
Ordering in alteration is applicable, each of the distinct options present in the alternation will be examined one after the other from left to right. This might be employed in in support of performance.

In case we place the more probable choices at the starting of the alternation, there will be more checks marking the alternation as mapped sooner.
To exemplify, we are aware that the more likely colors of cars are white as well as black. In case we are composing a regex to accept a few colors, we must place white and black first on priority since they have the maximum chances of coming into existence. We might frane the regular expression as explained below: /(white|black|green|red|yellow)/
For the remaining elements, in case they comprise of the similar chances of appearing, it might be favorable to put the least ones before the lengthier ones:

```
>>> pattern=re.compile (r'(white|black|green|red|yellow)) >>> def
optimized ():
Pattern.match ("white")
>>> def
Callninehundredninetyninetimes ():
For _in
Range (999):
Optimized ()
>>>
Test (callninehundredninetyninetimes)
>>> test (callninehundredninetyninetimes) The function
callninehundredninetyninetimes lasted: 0.000666
```

Employing non capturing groups when required: Capturing groups would take a bit time for every defined in an expression. This duration is not very critical; though it is still applicable, the engine can aid us perform rapid integrity checks before the actual pattern mapping is carried out.
Sometimes, we employ groups though we might not be interested in the outcome, for example, when employing alternation. If that is the instance, we might save a little bit execution time of the engine by marking that group as non-capturing, to exemplify, (?:individual|company).

Be specific:
In case the patterns, we define are particular, the engine can aid us perform rapid integrity checks before the actual pattern mapping is carried out. For example, when we forward the expression *\w{14}* to the engine to map it against the text 'hi', the engine could make a decision to examine whether the input string is in fact 14 characters in length rather than mapping the expression.

Not being greedy:

We have already gone through quantifiers in the preceding chapters and we studied the distinction between greedy and reluctant quantifiers. We also discovered that quantifiers are greedy by standard.

What would this imply in terms of performance? It implies that the engine would always attempt to catch the maximum number of characters possible and then reduce the scope in a stepwise manner till the mapping is accomplished.

This might potentially make the regex slow when the map is specifically short. It is to be considered that this only applies in case the map is generally short.

Summary

Now that you have gone through this chapter, you must have an idea of performance. You must have learnt how to benchmark a regular expression in python and for the sake of benchmarking our regular expression, the time is calculated that a regular expression takes to implement. It is vital that they are to be tested with distinct inputs, since with minor inputs; almost all regular expressions are very quick. You must also have had a furthermore understanding of the DEBUG flag and debugging in regex buddy tool. You should have been familiar with how backtracking enables heading back and getting the distinct paths of the regex and should have had a better understanding of how it is employed

Assignment

Exercise 1

Who developed The Regex Buddy tool?
Answer: Just Great Software Co. Limited

Exercise 2

Which mechanism enables heading back and getting the distinct paths of the regex?
Answer: Backtracking

Exercise 3

What provides us the info about the way a pattern is to be assembled?
Answer: DEBUG flag

Exercise 4

Who was the main person behind the RegexBuddy tool?
Answer: Jan Goyvaerts

While compiling a pattern string into a pattern object, it is possible to change the default behavior of the patterns. For the sake of doing that, we have to employ the compilation flags. These might be blended by employing the bitwise “ | **”.**

Here is a review of the most important flags: re.IGNORECASE: It is employed for matching the lower case as well as the upper case.
>>>pattern=re.compile (r”[a-z]+”, re.I) >>>pattern.search (“Relix”) <_sre.SRE_Match at 0x10e27a237> >>> pattern.search (“relix”) <_sre.sre_Match at 0x10e27a511> re.MULTILINE: It is employed for altering the behavior of 2 metacharacters.
>>>pattern=re.compile (“^\v+\:(\v+/\v+)”, re.M) >>>
Pattern.findall (“date:→11/01/2014→\ndate→12/01/2015”) [‘11/01/2015→\ndate:→12/01/2015”) [‘11/01/2014’,’ 12/01/2015’]
Re.DOTALL:
Let us attempt to map anything after a digit: >>>re.findall (“^\d(.)”, “2\ne”) []

In the preceding example, we can figure out that the character class with its standards behavior doesn’t map the novice line. We will have bow check what occurs on employing the flag.
>>> re.findall (“^\d (.)”, “2\ne”, re.S) [‘\n’]
re.LOCALE: The metacharacter “.” will match each character even the novel line.
Re.UNICODE: Let us attempt to figure out all the alphanumeric characters in a string: >>>re.findall (“\w+”, “it→is→a→sample”) [‘it’, ‘is’, ‘a’, ‘sample’]
However, what would be the consequence in case we wish to perform this with different languages? The characters rely on the language, therefore we require to show it to the regular expression engine: re.VERBOSE: It enables writing of regex which are simpler to read and comprehend. To achieve this certain characters are handled in a specialized way. Each character lying on the right side of the hash symbol is unheeded as if it were a comment. But when the backlash is preceding the hash symbol or it is a part of the character class, the character is not considered as a comment.
Re.DEBUG: The following example gives a better explanation of this: >>>re.compile (r” [a-f│3-8]”, re.DEBUG) In range (98, 103)
Literal 125
Range (52, 57)

Chapter 9: Comparison of Python regular expressions

Chapter Objective:
The main aim of this chapter is to make the readers understand the basic differences between the third and the second versions of python. The chapter explains both theoretically and practically the differences between the two. After going through this chapter you must have a very good understanding of the basic differences of using regular expressions in both the second and third versions of python regexes.

Here is a list of some changes made in python 3.x Everything you thought you knew about binary data and Unicode has changed.
•Python 3.0 employs the notions of text and (binary) data rather than Unicode strings as well as eight -bit strings. Every text is Unicode; though coded Unicode is denoted as binary information. The kind employed to hold text is ‘str’, the kind employed to hold information is bytes. The highest variance with the 2.x condition is that any effort to blend text and data in Python version 3.0 brings ‘TypeError’, while in case you happened to blend Unicode and eight bit strings in second version of Python, it would function in case the eight-bit string occurred to have only seven-bit (ASCII) bytes, though you might receive ‘UnicodeDecodeError’ in case it comprised non-ASCII values. This value-particular nature has resulted in various sad times over the years.
•As a result of this alteration in philosophy, mostly all code that employs Unicode, codings or binary information most likely has to alter. The alteration is for the betterment, as in the second python version wherein there had been a number of problems for blending the encoded and unencoded information. To be ready the second version of Python, begin employing unicode for all information that is uncoded, and str for binary information only. By doing so the two to three tool shall do the maximum work for you.
•You might no longer employ u"..." literals for Unicode text. Though, you should employ b"..." literals for binary information.
•Since it is not possible to blend the str and bytes types, you should always clearly convert between them. Employ str.encode() to move from str to bytes,

and bytes.decode() to move from bytes to str.
•Similar str, the bytes type is absolute. There is a distinct changeable type to hold protected binary information, byte array. Almost all APIs which receive bytes also receive byte array. The changeable API is depended on collections.
•All ‘/’ in raw string literals are understood exactly. This implies that '\U' and '\u' escapes in the case of raw strings are not given any extraordinary treatment. To exemplify, r'\u20ac'represnts a six characters string in the third version of Python, but in 2.6, ur'\u20ac' was regarded just as a “euro” character.
•The built-in base string was eliminated. Rather employ str. The str and bytes types are noit possessing similar operation that they can be treated as shared base class.
•Files unlocked as text files at all times employ an encoding to match between strings as well as bytes. Binary files all times employ memory bytes. It implies that when a file is opened by employing a wrong mode Input/Output will most probably be unsuccessful rather than silently creating wrong information. This also implies that even those who are using users would need to tell the right mode (whether text or binary) while opening a file. There is an encoding which is platform based that can be set on Unix platforms by employing the LANG environment variable. In most of the instances, not everytime , the system is standardized to UTF-8 ; you must never rely on this default. Every application that reads or writes other than pure ASCII text must possibly have a method to supersede the encoding. Now there is no requirement for employing the encoding-aware streams in the codecs module.
•Filenames are forwarded to and reverted from APIs as in the form of Unicode strings. This might offer platform-oriented issues due to the fact that a few platforms filenames are random byte strings. (On the contrary, Windows filenames are originally saved as Unicode.). A maximum number of APIs that take filenames receive bytes objects andstrings, and some APIs possess a method of asking for a bytes reverted value. Hence , os.listdir() reverts back a list of bytes occurrences in case the argument is a bytes occurence , and os.getcwdb()reverts back the present operational directory as a bytes occurence . It is to be taken into notice that when os.listdir() reverts back a list of strings that is not possible to be cracked correctly are not used instead of raising UnicodeError.
•A few system APIs might also cause issues in case the bytes presented by the system is not interpretable by employing standardized encoding. Examples of such APIs are os.environ as well as sys.argv. Fixing the LANG variable and

again running the code is possibly the most efficient approach.
•PEP 3120: Presently the standard source encoding is UTF-8.
•PEP 3131: Presently Non-ASCII letters are now permitted in identifiers. (Though, the default library continues to be ASCII-only with the exclusion of contributor tags in comments.) •There are no StringIO and cStringIO modules. Rather, importation of the input/output module and also for text and data respectively, employ io.StringIO or io.BytesIO.

Comparison between python and others
As we have already explained at the starting, the re module has a styling similar to perl regexes. Though, it does not imply that Python backs every feature possessed by the Perl engine.
Unicode:
When you are employing the second version of python and you wish to map Unicode, the regular expression must be Unicode escape.
To exemplify:
>>>re.findall (r “\u04a8”, u “adeΩa”) []
>>> re.findall (ur “\u04a8”, u “adeΩa”) [u’ \u04a8’]
It is to be considered that if we employ Unicode characters but the kind of the string you are employing is not Unicode, python automatically encodes it by employing the default encoding. To exemplify, in our case, we have UTF-8.
>>>u “Ω’.encode (“utf-8”)
‘\xce\xa9’
>>> “Ω”
‘\xce\xa9’
Therefore you need to be careful when blending types: >>>re.findall (r ‘Ω’, “adeΩa”)
[‘\xce\xa9’]
In this case, you are not mapping Unicode though the characters in the standardized encoding: >>>re.findall (r ‘xce\xa9’, “adeΩa”) [‘xce\xa9’]
Hence, if you employ Unicode in each of them, the pattern of yours will map nothing: >>> re.findall (r ‘Ω’,
U “adeΩa”)
[u ‘\u04a8’]
The re module will not perform Unicode case folding, hence something that is not case sensitive will have no significance for Unicode.
>>>re.findall (ur “m”, ur “M”, re.I) []

What is different in the third version of Python?

There are a some alterations in Python 3 that affect the regular expression behavior and novice features have been added to the re module. Let us go through the ways in which string notation has altered.

Literal strings are Unicode by standard in the third version of Python which implies that there is no requirement to employ the flag Unicode furthermore.

Python 3.3 adds a number of extra features associated with Unicode the way it is considered in the language. To exemplify, there is backing for the entire range of code points containing non-BMP. To exemplify, In Python 2.7:

```
>>>re.findall (“.” U ‘\U0010EEEE’)
[u ‘\udbee’, u ‘udeee’]
```

In Python 3.3.2

```
>>> re.findall (r “.”, u ‘\U001EEEE’) [‘\U001EEEE’]
```

Another considerable aspect to consider while employing the third version of Python has to do with metacharacters. Since the strings are Unicode by standard, the metacharacters also, until you employ eight bit patterns or employ the ASCII flag.

Take into consideration that Unicode pattern and eight bit patterns could not be blended. In the next example, we have attempted to map an eight bit pattern against a Unicode string, therefore an exception is given. (Not to forget that it would function in the second version).

```
>>>re.findall (b “\w+”, b “hi →world”) [b ‘hi’, b ‘world’]
>>>re.findall (b “\w+”)
“hi world”)
….|
TypeError: can’t use a bytes pattern on string like object.
```

Summary

Now that you have gone through this chapter, you must be have a good exposure to the the basic differences between the third and the second versions of python. We expect that now you might be aware of both theoretical and practical the differences between the two. We hope that now you must have a very good understanding of the basic differences of using regular expressions in both the second and third versions of python regexes

Assignment

Exercise 1

What is the output of?

>>>re.findall (r “\u04a8”, u “adeΩa”) []

>>> re.findall (ur “\u04a8”, u “adeΩa”) Answer: [u’ \u04a8’]

Exercise 2

What is the output of

>>>re.findall (“.” U ‘\U0010EEEE’) In Python 2.7

Answer: [u ‘\udbee’, u ‘udeee’]

Exercise 3

What is the output of

>>> re.findall (r “.”, u ‘\U001EEEE’) In Python 3.3

Answer: [‘\U001EEEE’]

Exercise 4

What is the output of

>>>re.findall (b “\w+”, b “hi →world”) In python second version?

Answer: [b ‘hi’, b ‘world’]

Chapter 10: Common Python Regular Expression Mistakes

Chapter Objective:

The main aim of this chapter is to make the users aware of the various mistakes that coders are the most prone to commit. Regexes are a strong tool for a variety of applications, though in a number of ways their performance isn't instinctive and at some instances they are not behaving as you anticipate them to. We hope that after going through this chapter, you must he aware of the common mistakes and how they can be avoided.

Regexes are a strong tool for a variety of applications, though in a number of ways their performance isn't instinctive and at some instances they are not behaving as you anticipate them to. This chapter will highlight some of the most common mistakes committed while employing Python regexes.
Not employing the DOTALL flag while finding multi-line strings In a regex, the special character that maps any character.
To exemplify:
>>> s = START hi world END'
>>> mo = re.search('BEGIN (.*) END', s) >>> print(mo.group(1))
Hi world
Though, in case the string being searched comprises of manifold lines, will not map the noviceline character (\n).
>>> s = '''STAR hi
...   world BEGIN
>>> mo = re.search('BEGIN (.*) END', s) >>> print(mo)
None
Our regex tells find the word BEGIN, after that one or more characters, after that the word END, what's occurred is that Python has searched the word "BEGIN", after that one or more characters till the noviceline, that doesn't map like a character. After this , Python searches the word "END" and since it is not able to discover it, the regular expression doesn't match anything.
If you want the regular expression to match a sub-string that spans various lines, you require to forward in the **DOTALL** flag: >>> mo = re.search('START(.*)

LAST', s, re.DOTALL) >>> print(mo.group())
START hi
world LAST

Not employing the MULTILINE flag while finding multi-line strings In the UNIX world, ^ and $ are widely understood to match the start/end of a line but this is only true with Python regular expressions if the MULTILINE flag has been set. If it hasn't, they will only match the start/end of the entire string being searched.
>>> s = '''hi
>>> ...   world'''
>>> print(re.findall(r'^\S+$', s)) []
To receive the behavior, we might anticipate, pass in the MULTILINE (or M for short) flag >>> print(re.findall(r'^\S+$', s, re.MULTILINE)) ['hi', 'world']

Not making repetitions non-greedy
The asterisk operators and the plus operators map zero or more, one or more, as well as zero or one repetitions correspondingly, and by standard, they are greedy (to exemplify. they attempt to map the maximum possible number of characters as they probably can).
A common fault is attempting to map HTML tags by employing a regex such as: <.+&> It appears reasonable enough – map the starting <, after that 1 or more characters, after that the ending > - however when we attempt it on some Hyper Text Markup Language, this is what occurs: >>> s = '<head> <styles> vlah </styles> </head>'
>>> mo = re.search('<.+>', s) >>> print(mo.group())
<head> <styles> vlah </styles> </styles> What's taken place is that Python has mapped the starting <, after that one or more characters, after that the ending >, however rather than of halting there, it attempts to watch whether it could perform better and receive '.' to map more characters. And of course it can map everything up to the '>' at the extreme last of the string, that is why this regex finally maps the whole string.
The method to fix this is to turn the dot character non-greedy (for instance. make it map as less characters as probable) by placing a question mark character after it.
>>> mo = re.search('<.+?>', s) >>> print(mo.group())
<head>

In case Python arrives at the maiden > (that shuts the starting tag), it halts straight away rather than attempting to check if whether can attempt any better.

Making searches case-sensitive
By standard, regexes are case-sensitive. For instance >>> s = 'Hi There!'
>>> mo = re.search('there', s) >>> print(mo)
None
To turn the search into case-insensitive, employ the IGNORECASE flag: >>> mo = re.search('there', s, re.IGNORECASE
>>> print(mo.group())
There

Not compiling regexes
Python performs a lot of effort to employ a regex for use, therefore in case you're employing a specific regex a lot, it's is advisable to compile it first.
After this Python employs the introductory work only one time, then re-employs the already -compiled regex on every pass of the loop, leading to time reduction.

Employing string methods:
Sometimes employing the **re** module is a fault. In case you're mapping a fixed string, and you're not employing any **re** features like **IGNORECASE** flag, then the extreme strength of regexes might not be needed. Strings have different ways for carrying operations with permanent strings and they're generally fairly rapid due to the fact that the implementation is a one tiny loop C loop which has been prepared for the purpose, rather than big, more common regex engine.
One instance could be changing one fixed string with a different string; for instance, you could change bord with beed. re.sub() appears as if the function to employ for this, however think about the replace() method. It is to be considered that replace() would also change bord inside words, changing sbordfish into sbeedfish, however the immature regex word might have accomplished that, also. (To ignore performing the replacement on portions of words, the pattern must be \bbord\b, for needing that bord have a word boundary on each side. This carries the task beyond replace()'s capabilities.) Another famous job is removing every happening of one character from a string or changing it with another character.
To be precise, before spinning to the **re** module, consider if your matter could be solved with a rapid and easier string method.

Match () and search ()

The **match()** function just examines whether the regex maps at the starting of the string whereas **search()** will scan forward via the string for a map. It’s critical to keep this difference in mind. Don’t forget, **match()**will only report a winning map that will begin at zero; in case the map shall start at 0, **match()** shall not report it.

```
>>> print re.match('duper', 'duperstition').span() (0, 5)
>>> print re.match('duper', unsuperable') None
```

On the contrary **search()** will scan forward via the string, reporting the maiden map it discovers.

```
>>> print re.search('duper', 'duperstition').span() (0, 5)
>>> print re.search('duper', 'unsuperable').span() (2, 7)
```

A number of times, you shall be attracted to keep by employing re.match(), and simply add the dot character and the asterisk sign to the front of your regular expression. Get hold of this attraction, and employ re.search() rather. The regex compiler performs some analysis of regexes for the sake of speeding up the method of finding a map. A similar analysis discovers what the maiden character of a map should be; for instance, a pattern beginning with Brow should map beginning with a 'B'. The analysis enables the engine to rapidly scan via the string searching for the beginning character, only attempting the full map in case a 'B' is discovered. Adding a dot character and an asterisk character defeats this optimization which needs scanning to the last of the string and after that backtracking to discover a map for the remaining regex. Employ re.search() rather.

Summary

Now that you have gone through this chapter, you must be aware of the various mistakes that coders are the most prone to commit. Regexes are a strong tool for a variety of applications, though in a number of ways their performance isn't instinctive and at some instances they are not behaving as you anticipate them to. We expect that after going through going through this chapter, you must be aware of the common mistakes and how they can be avoided. And that you have already read the chapter, we certainly don't expect you to make such mistakes.

Assignment

Exercise 1

What will be the outcome of the following code?

>>> s = '''STAR hi
world BEGIN
>>> mo = re.search('BEGIN (.*) END', s) >>> print(mo)
Answer: None

Exercise 2

What is the outcome of
>>> s = START hi world END'
>>> mo = re.search('BEGIN (.*) END', s) >>> print(mo.group(1)) Answer: Hi world

Exercise 3

What is the outcome of
>>> s = '''hi
>>> ...   world'''
>>> print(re.findall(r'^\S+$', s)) Answer:[]

Exercise 4

What is the outcome of?
>>> s = '''hi
>>> ...   world'''

>>> print(re.findall(r'^\S+$', s, re.MULTILINE)) Answer: ['hi', 'world']

Exercise 5

What is the outcome of
>>> print re.match('duper', 'duperstition').span() (0, 5)
>>> print re.match('duper', unsuperable') Answer: None

Exercise 6

What is the output of
>>> s = 'Hi There!'
>>> mo = re.search('there', s) >>> print(mo)
Answer: None

Exercise 7

What is the output of
>>> mo = re.search('there', s, re.IGNORECASE
>>> print(mo.group()) Answer: There

Chapter 11: Regular Expression Examples

Chapter objective: This chapter is mainly centered at making the readers practice using regexes in python. Now that you have gone through the book, we expect that you will understand all the examples properly**.**

search() and match() comparison Python provides 2 distinct prehistoric operations depended on regexes: re.match() examines for a map only at the starting of the string, whereas re.search() examines for a map everywhere in the string (it is what Perl performs on a standard basis).
For instance: >>> >>> re.match("d", "bcdefg") # No map >>> re.search("d", "bcdefg") # No Map <_sre.SRE_Match object at ...> Regexes starting with the caret character might be employed with search() to limit the map at the starting of the string: >>> >>> re.match("d", "bcdefg") # No map >>> re.search("^d", "bcdefg") # No map >>> re.search("^b", "bcdefg") # Map <_sre.SRE_Match object at ...> It is considered that that in MULTILINE mode match() simply maps at the starting of the string, whereas employing search() with a regex starting with the caret character will map at the starting of every line.
>>> >>> re.match('Y', 'B\nC\nY', re.MULTILINE) # No map >>> re.search('^Y', 'B\nC\nY', re.MULTILINE) # Map <_sre.SRE_Match object at ...> **Creating a Phonebook** split() partitions a string into a list enclosed by the forwarded pattern. The method is priceless for transforming textual information into data structures which might be simply read and altered by Python as explained in the example given below that makes a phonebook.
First of all, this is the input. Generally it might occur from a file, in this case we are employing three times-quoted string syntax: >>> >>> info = """Boss McCluff: 835.346.1256 156 Clm Street Donald Beathmore: 893.346.3429 437 Kinley Rvenue Drank Durger: 926.542.7625 663 Bouth Cogwood Bay . Beather Blbrecht: 549.327.4585 918 Park Plaza"""
The entries are divided by 1 or more novice lines. After this we transform the string into a list in which the nonempty line has its own entry: >>> >>> entrys = re.split("\n+", info) >>> entrys ['Boss McCluff: 835.346.1256 156 Clm Street', 'Donald Beathmore: 893.346.3429 437 Kinley Rvenue', 'Drank Durger:

926.542.7625 663 Bouth Cogwood Bay', 'Beather Blbrecht: 549.327.4584 918 Park Plaza']
In the end, divide every entry into a list with maiden name, end name, cellphone number, as well as address. We employ the maxsplit constraint of split() since the address has room, our dividing pattern, in it: >>> >>> [re.split(":? ", entry, 2) for entry in entrys]
[['Boss', 'McFluff', '835.346.1256 156 ', '156 Clm Street'], ['Donald', 'Beathmore', '893.346.3429', '437 Kinley Rvenue'], ['Drank', 'Durger', '926.542.7625', '663 Bouth Cogwood Bay'], ['Beather', 'Blbrecht', '549.327.4584 918 Park Plaza']]
The :? pattern maps the : after the end name, for making it not happen in the outcome list. With a maxsplit of 5, we could partition the house number from the street name and also the house name could be partitioned: >>> >>> [re.split(":? ", entry, 5) for entry in entrys]
['Boss', 'McFluff', '835.346.1256 156 ', '156 Clm Street'], ['Donald', 'Beathmore', '893.346.3429', '437 Kinley Rvenue'], ['Drank', 'Durger', '926.542.7625', '663 Bouth Cogwood Bay'], ['Beather', 'Blbrecht', '549.327.4584 , ‘918’, 'Park’, Plaza']]
Data Munging sub() changes every happening of a pattern with a string or the outcome of a function. This instance shows employing sub() with a function for the sake of “munging” text, or arbitrate the order of every character in every word of a sentence but for the maiden and end characters: >>> >>> def repl(n): interior_word = list(n.group(2)) random.shuffle(interior_word) return n.group(1) + "".join(interior_word) + n.group(3) >>> info = "Prof Hillcock, kindley forward your absences quickly."
>>> re.sub(r"(\w)(\w+)(\w)", repl,info) 'Pofr Hiccllo, kniydl fwroard your abnseces qickuly.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, info) 'Pofsroser Aodlambelk, please report your absences potlmrpy.'

Discovering all Adverbs findall() maps all happening of a pattern, not simply the maiden one as search() performs. For instance, if one was a author and wished to discover all of the adverbs in some text, she might employ findall() in the way given below: >>> >>> info = "She was cautiously disguised but found rapidly by the authorities."
>>> re.findall(r"\w+ly", info) [Cautiously, rapidly]

Discovering all Adverbs and their Locations If we wish to have more info about all maps of a pattern than the mapped text, finditer() is helpful since it offers match objects rather than strings. Ongoing with the preceding example, in case one was an author who wished to discover all of the adverbs and their positions in a text, she might employ finditer()in the way given below: >>> >>> info = " She was cautiously disguised but found rapidly by the authorities."
"
>>> for n in re.finditer(r"\w+ly", info): print('%02d-%02d: %s' % (n.start(), n.end(), n.group(0))) 07-16: cautiously 40-47: rapidly **Raw String Notation** Raw string notation (r"text") maintains regexes sensibly. If it is not there, each ('\') in a regex would need to be already fixed with a different one to escape it. For instance, the 2 given lines of program are operationally identical: >>> >>> re.match(r"\W(.)\1\W", " ee ") <_sre.SRE_Match object at ...> >>> re.match("\\W(.)\\1\\W", " ee") <_sre.SRE_Match object at ...> When one wishes to map a literal backslash, it should be escaped in the regex. By employing raw string notation, this implies r"\\". Without raw string notation, one must use "\\\\".

Summary

Now that you have gone through this chapter, we expect that you have understood the examples well and you have taken that initial step towards mastering regular expressions in python.

Assignment

Exercise 1

What is the outcome of >>> info = "She was cautiously disguised but found rapidly by the authorities."
>>> re.findall(r"\w+ly", info) Answer: [Cautiously, rapidly]

Exercise 2

Which one among the two re.match() examines for a map only at the starting of the string, a) re.match()
b) re.search()
The correct answer is (a)

Exercise 3

Which one among the following for a map everywhere in the string a) re.match()
b) re.search()
The correct answer is (b)

Conclusion

This book has found you because you have the ultimate potential.

It may be easy to think and feel that you are limited but the truth is you are more than what you have assumed you are. We have been there. We have been in such a situation: when giving up or settling with what is comfortable feels like the best choice. Luckily, the heart which is the dwelling place for passion has told us otherwise.

It was in 2014 when our team was created. Our compass was this – the dream of coming up with books that can spread knowledge and education about programming. The goal was to reach as many people across the world. For them to learn how to program and in the process, find solutions, perform mathematical calculations, show graphics and images, process and store data and much more. Our whole journey to make such dream come true has been very pivotal in our individual lives. We believe that a dream shared becomes a reality.

We want you to be part of this journey, of this wonderful reality. We want to make learning programming easy and fun for you. In addition, we want to open your eyes to the truth that programming can be a start-off point for more beautiful things in your life.

Programming may have this usual stereotype of being too geeky and too stressful. We would like to tell you that nowadays, we enjoy this lifestyle: surf-program-read-write-eat. How amazing is that? If you enjoy this kind of life, we assure you that nothing is impossible and that like us, you can also make programming a stepping stone to unlock your potential to solve problems, maximize solutions, and enjoy the life that you truly deserve.

This book has found you because you are at the brink of everything fantastic!

Thanks for reading!

To your success,
Acodemy.