

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT Chapter 11: Regular Expressions

Regular Expressions – Learning Objectives

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- LO 1 Realise the **need** for regular expressions (RE) for searching/matching
- LO 2 Understand the concept of **special characters** or **meta-characters** in RE
- LO 3 Use **start of string anchors** and **end of string anchors**
- LO 4 Explain the two distinct uses of the re module namely (i) as re.compile() (ii) as re module convenience functions
- LO 5 Differentiate between pattern objects and match objects
- LO 6 Employ the re.match(), re.search() , re.findall() and re.sub() methods
- LO 7 Appreciate that in the regular expression module, there are functions of the re module and methods of the Pattern object which may have the same name but different **signatures**
Example, there is a re.search() function and also a Pattern.search() method
- LO 8 Use the group(), start() end() and span() methods of the match object
- LO 9 Differentiate between greedy and non-greedy matching.

Introduction

- **A Regular Expression (RE) is some special sequence of characters, which helps you to match other strings. A regular expression may be a simple string containing ordinary characters or may contain special characters or meta characters. The Python re module provides regular expression support.**
- **Regular expressions are also called REs, or regexes, or regex patterns. REs are used to match patterns.**

Using REs, you can do the following:

- **Given a pattern, find out if it occurs in a given string.**
- **Given a string, find out if it matches a given pattern.**
- **You can also use REs to modify a string or to split it in various ways.**

Raw strings in Python

- The Python parser interprets `'\'` (backslashes) as escape characters in literal strings.
- Therefore, if you use a special sequence, such as `\n` inside a literal string, then the Python parser will recognize it as a newline character and `\n` will be replaced by a new line.
- However, there may be instances where you do not want to replace `\n` with the newline character. That is when the raw string comes into the picture.
- If you add the character `r` before a normal Python string, then the Python interpreter treats this string as a raw string.
- The `r` in front of the string tells the Python interpreter not to make any replacements/ substitutions within the string.
- Therefore, by putting an `r` and converting a normal string to raw string, you tell the Python interpreter to leave your string alone.

Definition of Regular Expression (RE)

A regular expression (also called RE or regex) is simply some means to write down a pattern describing some text. Regexes are used for four main purposes as follows:

- **Validation or Checking:** See whether a part of the text fulfills some criteria. (For instance, if it is an email id then does it have the @ character in it?)
- **Searching:** Searching can be for a fixed string or for a string which may have variable forms. For instance, in British English it is colour but in American English it is color.
- **Searching and replacing**
- **Splitting strings**

Special characters, groups of characters and anchors

The first step in understanding regular expressions is understanding the following three things:

- **Understanding special characters in forming patterns and**
- **Matching groups of characters**
- **Anchors**

Each of these are discussed in subsequent slides.

“Ordinary” and “Special” characters.

What are ordinary characters?

- **Characters, such as ‘a’ or ‘A’ or ‘1’ or any combination of them are ordinary characters.**
- **Regular expressions can be created using ordinary characters. Such regular expressions will match only themselves.**
- **For instance, a regular expression containing ordinary characters `boa` will match ‘board’, ‘boat’, ‘boast’, and so on.**

What are special meta characters?

- **Some characters, such as ‘|’ or ‘(’, are special. Special characters either:**
 - **represent classes of ordinary characters;**
 - **Or change the behavior of other characters around them.**
- **Further, RE can be used to match either individual characters or a group of characters.**

Meta characters

RE can be used to match either individual characters or a group of characters.

In Python, the following is the complete list of meta characters or special characters used to form a pattern (Since these characters are special, they don't match themselves but rather represent some other action) as follows:

. ^ \$ * + ? { } [] \ | ()

[and]. (Square brackets)

The square brackets together represent a character class, that is, they stand for a set of characters you want to match.

You have two options when using a set of square brackets:

- (i) You may enclose individual characters inside them. For instance, `[abc]` represents the three characters a, b and c.
- (ii) You may indicate a range of characters using hyphen, that is, `(-)`. For instance, `[a-z]` would represent lower case alphabet.

How character class matches a pattern can be best understood by the following example: Suppose you have a regular expression of type `a[bc][de]f`. Then, it will match `abdf`, `acdf`, `abef` and `acef`.

^ (Also called caret)

Note that “character class” means “square brackets”. Further note that ^ is used in two different ways:

- 1. Inside a character class: You can match those characters which are not listed within the class by complementing the given set of characters. This can be done by putting a '^' as the first character of the class. For instance, [^aeiou] means not a vowel (that is, anything which is not a vowel). Similarly, [^0-9] will match a non-numeric string like cat but will not match a numeric string, such as '00123'**
- 2. '^' outside a character class, matches at the beginning of the line. Hence '^Tom' will match 'Tom' went but will not match 'Please call Tom' because the second string does not begin with Tom. (Note: Outside the character class '^' is very similar to '\A').**

Backslash (\)

Backslash in RE has two distinct uses:

- 1. It is used to escape all the meta-characters. This situation will arise when you want to match a meta character and so you don't want to treat the meta character like a special character, but rather, you want to treat a meta-character like an ordinary character. For instance, if you want to match a backslash (\), then you can precede it with another backslash, that is, \\. The first backslash indicates to the Python interpreter that the second backslash will not have special meaning, but have an ordinary meaning of a backslash. So the first backslash “escapes” the second backslash.**
- 2. In Python, there are some special sequences, which begin with a backslash (\) and indicate some special meaning. For instance, \d indicates any of the 10 digits from 0 to 9.**

Some commonly used regular expressions with backslash (\)

<code>\s</code>	<code>\s</code> will match any white space characters including spaces, tabs, newline, and so on. So you can think of <code>\s</code> as a Unicode separator . <code>\s</code> is equivalent to character class <code>[\t\n\r\f\v]</code> . Note that <code>\f</code> stands for form-feed . <code>\v</code> stands for vertical tab. <code>\r</code> stands for carriage return
<code>\S</code>	<code>\S</code> is opposite of <code>\s</code> . So <code>\S</code> will match anything (that is any character) which is not a white space or which is not a Unicode “separator”. So <code>\S</code> is equivalent to the character class <code>[^\t\n\r\f\v]</code> .
<code>\d</code>	<code>\d</code> will match any of the 10 decimal digits from 0 to 9. So, it is equivalent to the character class <code>[0-9]</code> .
<code>\D</code>	<code>\D</code> is opposite of <code>\d</code> . So, it stands for non-digit characters. <code>\D</code> is equivalent to the character class <code>[^0-9]</code> .
<code>\w</code>	<code>\w</code> stands for word character. It includes alphabet (Both small and big case, digits and underscore, that is, <code>(_)</code>). So <code>\w</code> is equivalent to the character class <code>[a-zA-Z0-9_]</code> .
<code>\W</code>	<code>\W</code> is opposite of <code>\w</code> . So it will match a non-alphanumeric character. It will also not match underscore, that is, <code>(_)</code> . So <code>\W</code> is equivalent to the class <code>[^a-zA-Z0-9_]</code> .

'|'is alternation and '.'(Dot)

- '| is alternation, or the “or” operator. If X and Y are regular expressions, then $X|Y$ will match any string that matches either X or Y.
- '.'(Dot.) The dot, that is, (.) matches any character except a newline.

Matching group of characters

Apart from the ability to match individual characters, the re module also has the capability to match varying sets of characters.

- ***** is used to specify that the previous character will be matched zero or more times.
- **+** It matches one or more times
- **?** The question mark character, that is, (?) will match either once or zero times. So, you can think of the question mark, that is, (?) as representing something that is optional.
- **{m}** stands for exactly m repetitions.
- **{m,n}** where m and n are integers. Here m indicates the minimum possible occurrences while n indicates the maximum number of occurrences of a character.

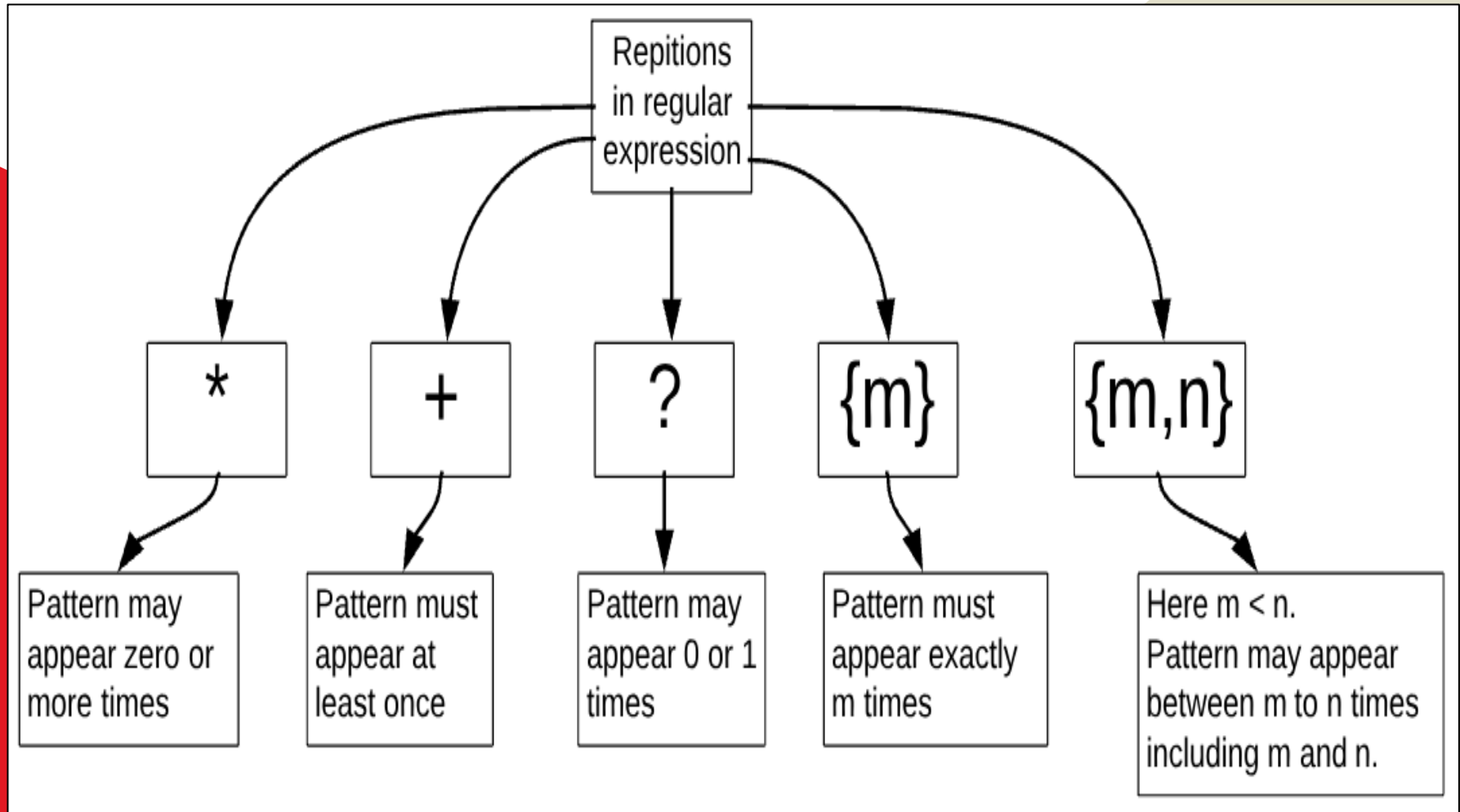
Various ways of using {m, n}

{0,} is the equivalent as *

{1,} is equivalent to +,

{0,1} is the same as ?.

Visual representation of the various ways of repetition in regular expressions



RE Anchors

It must be noted that in RE, anchors don't match any characters at all. They provide an anchor to the regexp to match at a certain position.

The following things can be said about anchors:-

- Anchors are regex tokens.**
- They in themselves don't match any character.**
- On the other hand, they “assert” something about the matching process. For example if you want to match at a “particular location” such as “beginning of a string”, or “beginning of a line”, then you can use “anchors” which in turn tell the regexp engine to search at such specified places only.**
- One advantage of using an “anchor” is that you can “narrow down” your match to “specified locations” only. For example suppose you want to match or find only those strings which “begin” with a digit but not those which “don't begin with a digit but can contain digits”. Here you can use an “anchor” to tell the regexp engine to search only at “beginning of a string” and not “inside a string”.**
- Another advantage of using anchors is that since you “narrow down” the search to specified locations only, it makes the match process faster.**

Common anchors

	pattern	Function
1	\A	\A matches only at the beginning of the string
2	\Z	\Z matches only the end of the string
3	\b	Represents word boundary
4	^	Matches start of a string. (Note as explained above, the ^ character has 2 behaviors. The character ^ acts as an anchor outside the square brackets.)
5	\$	Matches end of string or line

2 different ways of using the re module

In Python, the re module for RE can be used to look for the “some_pattern” in a given “some_string” in two distinct ways:-

1. **re.compile(some_pattern) and the Pattern Objects.** Here the re.compile(some_pattern) methods “returns” a Pattern object. So once you get this “Pattern object” using the re.compile(some_pattern) function, you can use the various methods and attributes of this object for doing searches and matches.
2. **re module convenience functions.** Here you do not create a “Pattern object”, rather you directly do the match using convenience functions. A number of “convenience functions” are provided. So you provide both the “some_pattern” as well as the “some_string” being searched to the convenience function as arguments and get your results.

Method 1:- Creating a Pattern object

Note here, that you will give only one parameter to `search(some_string)`, namely the string `str`.

So the method works in two steps.

- First, you create a Pattern object using only `some_pattern`
- Second, you use methods, such as `match(some_string)` or `search(some_string)` or other methods on this Pattern object you created.

Method 2:- re module convenience functions

Here you don't have to create a Pattern Object yourselves, but rather call the method of the Pattern object directly. Here, you have to give two attributes to the convenience functions.

The re module convenience functions are functions which take two parameters, namely

1. `some_pattern`
2. `some_string`.

Examples of these methods are `re.match(some_pattern, some_string)` where `some_pattern` is the pattern and `some_string` is the string on which the match is to be done.

Note that the names of the “convenience functions” namely `match()` or `search()` are the same as in previous case (Where pattern object was created) but the “convenience functions” here take two parameters and not one.

Pattern Object and Match object

The re module can create basically two types of objects:

(i)Pattern object and

(ii)Match object

Pattern object is created when a string is “compiled” using the function `re.compile(some_pattern)`, where `some_pattern` is some string.

You can create a Match object in 2 different ways. They are:-

1.Using functions of the re module. Here you use re functions of type `re.match(pat, string)`. The return value of this `re.match()` function is a match object. Note the `re.match()` function takes 2 parameters namely (1) `pat` ie the `pattern_to_be_searched` and (2) `string` ie the `string_to_be_looked_in`

2.Using methods of the Pattern object you created earlier using `re.compile(some_pattern)`. The return of this function is a Pattern object, which has methods to create a Match object. Suppose you get a Pattern object say `my_pat`. Then you can use a method of this Pattern object. For example you can use:- `my_pattern.match(string_to_be_looked_in)`. Here the `my_pattern.match()` method takes only 1 parameter namely the `string_to_be_looked_in`

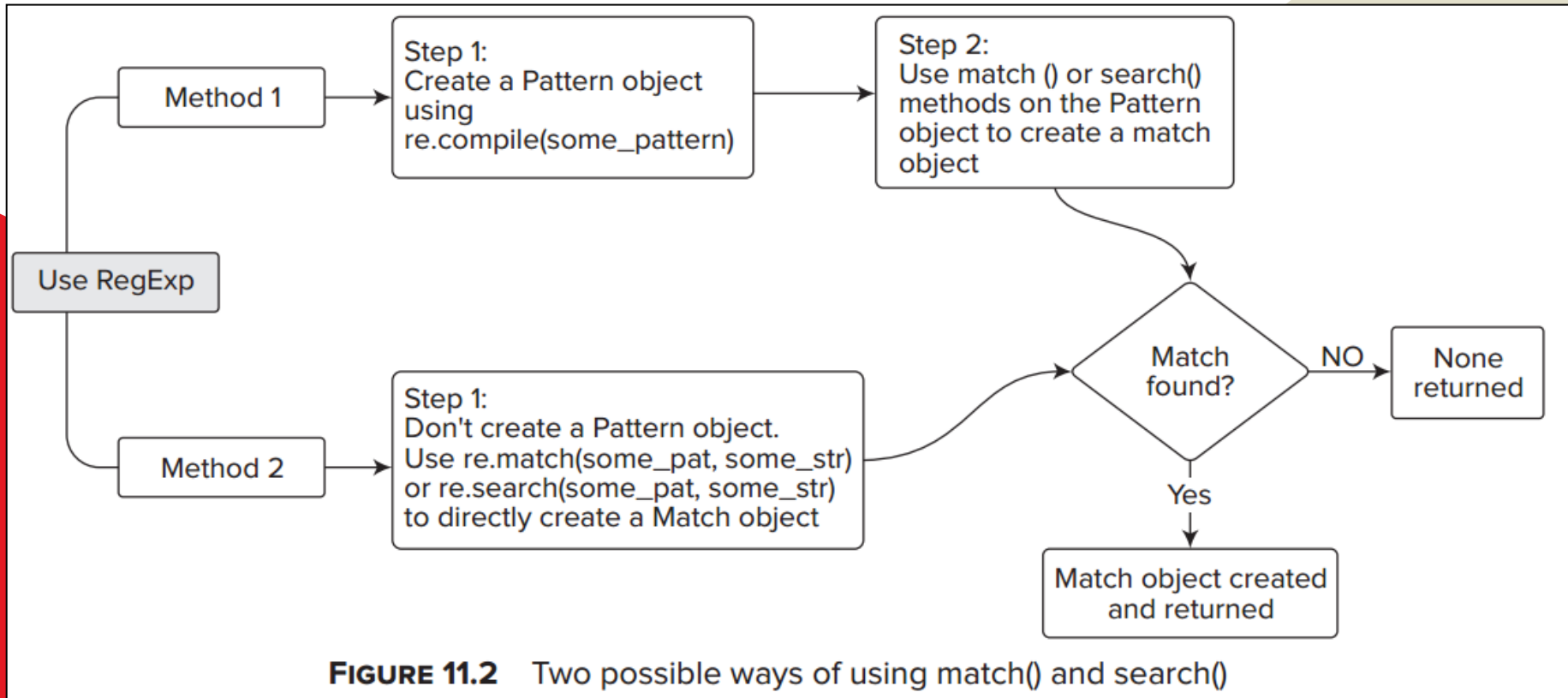
What is a Match object and why have it?

- The previous slide talked about creating a Match object.
- But what is a Match object and why do we need it?
- A Match object is an object which contains information about a match between a pattern and a string.
- If the pattern being searched is not found in the given string, then the Match object created by this “comparison” will be None. So None indicates no match.
- However if there is some match (Which may be a single or more than 1 match, ie a pattern occurs at more than 1 place in a string), then this Match object created will not be a None, but rather it will be an object. Now we can use the various methods and attributes of this Match object to know “details of the match” which took place.

Table showing summary of various functions of the re module used for matching

Method	Purpose
<code>match(pat, str)</code>	It checks whether the string str starts with the pattern pat . If pat is found in str , it returns a match object. Else it returns None.
<code>search(pat, str)</code>	It checks for occurrence of pattern pat in the string str . The pattern may be anywhere in the string str . If pat is found in str , it returns a match object. Else it returns None.
<code>findall(pat, str)</code>	It checks for all instances of pattern pat in the string str . Then it returns the found patterns in the form of a list. Therefore, this method does not return a match object.
<code>finditer(pat, str)</code>	It checks for instances of pattern pat in string str and returns an iterator, which is a match object.

Figure shows 2 ways of creating a Match object



Difference between match() and search() methods/functions.

- It is also important to know the difference between the `re.match()` and `re.search()` functions.
- The `match()` method only finds matches of the given pattern if they occur at the start of the string being searched.
- However, the `search()` method looks for a match not just at the beginning of the given string but looks for a match over the entire string.

Please see the examples given in 11.5.2, to clarify the concept.

Using group() method of the match object -- 1

1. Recall that the `match()` and the `search()` functions in Python RE return a Match object which has certain attributes/ methods of its own. One important attribute of the Match method is the `group()` method. Grouping is the capability to address certain subparts of the entire substring matched by the regex.
2. A group is that part of a regular expression pattern, which is bounded by parenthesis(). If you add a pair or pairs of parenthesis, to a regular expression pattern, then it will not change what that pattern matches, but it will form groups within the matched sequence.
3. If there is a single argument, that is, a single pair of parenthesis, then you will get a single string as result. However, if there are multiple arguments, then the result that you get is a tuple with one item for each argument. The `group()` method returns one or more subgroups of the match.

Using group() method of the match object -- 2

4. The possibilities are as follows:

- a. **group() or group(0):** Both are equivalent and return the entire matched substring in the form of a string.
- b. **group(N):** Returns the Nth matched subgroup in the form of a string. Note that the first subgroup is group(1) and not group(0).
- c. **group(...m, n, k, ..):** This is a group function with multiple arguments. This will return a tuple of the subgroups. The number of items in the tuple are the same as the number of arguments to the group() method. For instance, group(1,3,4) will return a tuple of three items, namely the 1st, 3rd and 4th subgroups of the matched substring.

Using group() method of the match object – 3 (Example code)

The example code in the book is reproduced below. It shows how the 3 different formats of the group() method of a Match object may be used.

(There is detailed explanation in the book)

```
1 >>>matchObj = re.match(r'(spo*n) (spo*n)', 'spn spon spoon spoon')
2 >>>matchObj.group()
3 'spn spon'
4 >>>matchObj.group(0)
5 'spn spon'
6 >>>matchObj.group(1)
7 'spn'
8 >>>matchObj.group(2)
9 'spon'
10 >>>matchObj.group(1,2)
11 ('spn', 'spon')
```

Using start() and end() methods of the Match object

- The general form of these methods is `start(group_number)` and `end(group_number)`.
- If you have a Match object say `matchObj`, then `matchObj.start(group_number)` will give the index of the substring matched by the group and `matchObj.end(group_number)` will give the index of the end of the substring matched by the group.
- Note that if `group_number` is not given, or if it is 0 then it will default to the entire matched substring.

The following script (From book) shows the use of `match_obj.start()` and `match_obj.end()` methods of a Match Object.

```
1 >>> import re
2 >>> myStr = '012xxxxx89'
3 >>> matchObj = re.search(r'x+', myStr)
4 >>> matchObj.group()
5 'xxxxx'
6 >>> matchObj.start()# Index of start ie first 'x' in '012xxxxx89' is 3
7 3
8 >>> matchObj.end()
9 8
10 >>> myNewStr = myStr[:matchObj.start()] + myStr[matchObj.end():] # All 'x'
11 are being removed
12 >>> myNewStr
13 '01289'
14 >>> matchObj.span()
15 (3, 8)
```

The span() method of the Match object

The span function returns a tuple, which contains the (start, end) positions of the match. Suppose you have a Match object, say matchObj. Then span() is equivalent to (matchObj.start(), matchObj.end()).

Note that span()[0] is the same as start() and span()[1] is the same as end(). The syntax span()[0] may look a bit unusual, but it is actually quite valid because remember that span() is a sequence and therefore, indexable.

```
1 import re
2 myP = re.compile('\d+')
3 myI = myP.finditer('10 Ten 9 Nine 8 Eight 7 Seven')
4 for match in myI:
5     print(match.span(), 'From index', match.span()[0], 'to index',
6           match.span()[1])
```

```
6 #Output is:
7 (0, 2) From index 0 to index 2
8 (7, 8) From index 7 to index 8
9 (14, 15) From index 14 to index 15
10 (22, 23) From index 22 to index 23
```

The above example code (From book) explains the use of **span()** method

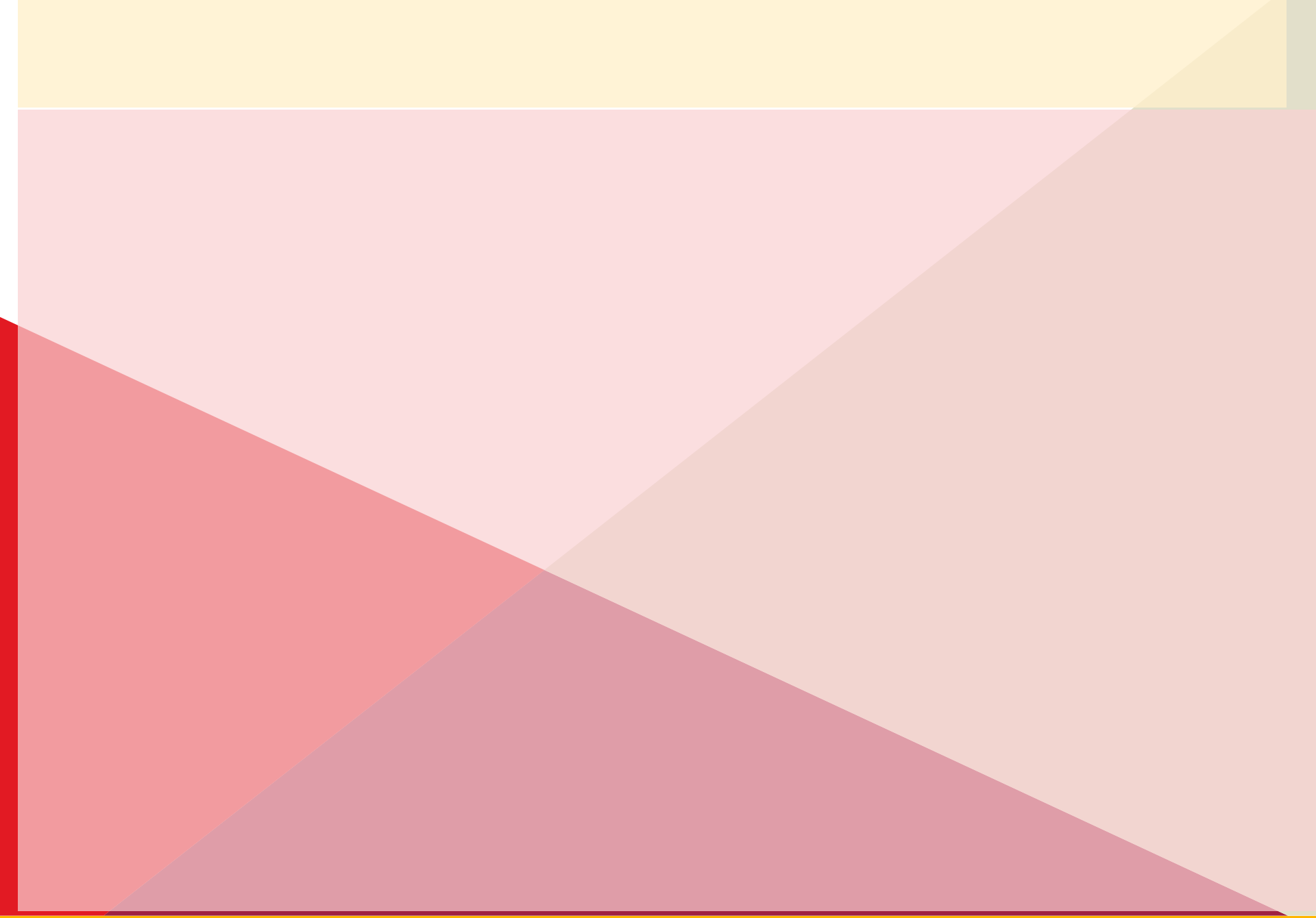
Greedy versus non-greedy matching

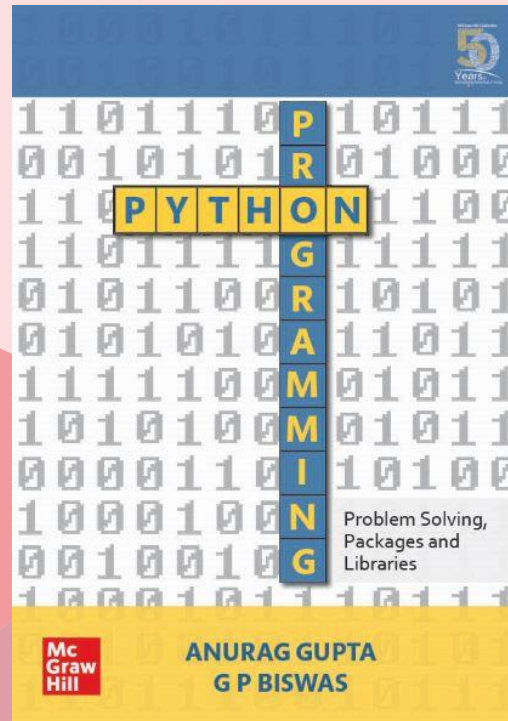
- In general, a qualifier like `*` will try to match as much of the string as possible.
- Same is the case with `{m, n}`. The default behaviour is that a special character will try to match as much of the search sequence (string) as possible.
- This default behaviour is called Greedy Match. It is the normal behaviour of a regular expression, but sometimes this behaviour is not desired.
- To do this you can use a qualifier, that is, question mark, that is, `?`. So the question mark, `(?)` is a greedy qualifier, that is, it qualifies a greedy behaviour into a non-greedy (also called lazy) behaviour.
- So expressions, such as `*?`, `+?`, `??` or `{m, n}?` will match as little of the string as possible.
- Note that the question mark ie `(?)` is a greedy qualifier, that is, it qualifies a greedy pattern into a non-greedy pattern.
- Note: The term greedy qualifier is slightly confusing. It means it qualifies or alters a greedy into a non-greedy pattern. So when you say that a question mark, that is, `(?)` is a greedy qualifier, it means that a greedy pattern has been qualified by the question mark, that is, `?` into a non-greedy one.

Greedy versus non-greedy matching -- Example

```
1 import re
2 str_html = '<!DOCTYPE html> <html> <head> </head> </html>'
3 print('length string->', len(str_html))
4 # 1-----GREEDY
5 pat_greedy = '<.*>'
6 print('length greedy->', re.match(pat_greedy, str_html).span())
7 print('groups greedy->', re.match(pat_greedy, str_html).group())
8 # 2-----NON-GREEDY
9 pat_nongreedy = '<.*?>' # has ? after * which makes it non-greedy
10 print('length non-greedy->', re.match(pat_nongreedy, str_html).span())
11 print('groups non-greedy->', re.match(pat_nongreedy, str_html).group())
12 # OUTPUT- - - - -
13 length string-> 45
14 length greedy-> (0, 45)
15 groups greedy-> <!DOCTYPE html> <html> <head> </head> </html>
16 length non-greedy-> (0, 15)
17 groups non-greedy-> <!DOCTYPE html>
```

The above example code (From book) shows the difference between a “greedy” and a “non-greedy” pattern.








Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

 support.india@mheducation.com

 1800-103-5875

 www.mheducation.co.in

in

