Introduction to regular expressions

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

What is a regular expression?

REGular EXpression or regex:

String containing a combination of normal characters and special metacharacters that describes patterns to find text or positions within a text

```
r'st\d\s\w{3,10}'
```

2. What is a regular expression?
A regular expression, or regex, is a string that contains a combination of normal and special characters that describes patterns to find text within a text.
In Python, the r at the beginning indicates a raw string. It is always advisable to use it.

What is a regular expression?

> **REGular EXpression or regex:**

> *String containing a combination of **normal characters** and special metacharacters that describes patterns to find text or positions within a text*

```
r'st\d\s\w{3,10}'
```

- Normal characters match themselves (`st`)

3. What is a regular expression?
We said that a regex contains normal characters, or literal characters we already know. The normal characters match themselves. In the case shown in our slide, st exactly matches an s followed by a t.

What is a regular expression?

> **REGular EXpression or regex:**

> *String containing a combination of normal characters and **special metacharacters** that describes patterns to find text or positions within a text*

`r'st\d\s\w{3,10}'`

- Metacharacters represent types of characters (`\d` , `\s` , `\w`) or ideas (`{3,10}`)

4. What is a regular expression?
They also contain special characters. Metacharacters represent types of characters. Let's look one by one as they appear in the slide. Backslash d represents a digit.

What is a regular expression?

REGular EXpression or regex:

String containing a combination of normal characters and ***special metacharacters*** *that describes patterns to find text or positions within a text*

```
r'st\d\s\w{3,10}'
```

- Metacharacters represent types of characters (`\d` , `\s` , `\w`) or ideas (`{3,10}`)

5. What is a regular expression?
backslash s a whitespace,
6. What is a regular expression?
backslash w a word character. They also represent ideas, such as location or quantity.

7. What is a regular expression?
In the example, 3 and 10 inside curly braces indicates that the character immediately to the left, in this case backslash w, should appear between 3 and 10 times.

What is a regular expression?

REGular EXpression or regex:

*String containing a combination of normal characters and **special metacharacters** that describes patterns to find text or positions within a text*

```
r'st\d\s\w{3,10}'
```

- Metacharacters represent types of characters (`\d` , `\s` , `\w`) or ideas (`{3,10}`)

8. What is a regular expression?
We said that regex describes a pattern. A pattern is a sequence of characters that maps to words or punctuation.

What is a regular expression?

REGular EXpression or regex:

String containing a combination of normal characters and ***special metacharacters*** *that describes patterns to find text or positions within a text*

```
r'st\d\s\w{3,10}'
```

- Metacharacters represent types of characters (`\d` , `\s` , `\w`) or ideas (`{3,10}`)

9. What is a regular expression?
As a data scientist, you will use pattern matching to find and replace specific text. To validate strings such as passwords or email addresses. Why use regex? They are very powerful and fast. They allow you to search complex patterns that would be very difficult to find otherwise.

What is a regular expression?

> **REGular EXpression or regex:**

> *String containing a combination of normal characters and special metacharacters that describes **patterns** to find text or positions within a text*

- Pattern: a sequence of characters that maps to words or punctuation

10. The re module
Python has a useful library, the re module, to handle regex. You can import it as shown in the code. Let's see how it works. To find all matches of a pattern, we use the dot findall method. It takes two arguments: the regex and the string. In the code, we want to find all the matches of hashtag movies in the specified string. The method returns a list with the two matches found.

What is a regular expression?

> **REGular EXpression or regex:**

> *String containing a combination of normal characters and special metacharacters that describes patterns* ***to find text or positions within a text***

- Pattern matching usage:
  - Find and replace text
  - Validate strings
- Very powerful and fast

The re module

```
import re
```

- Find all matches of a pattern:

re.findall(r"regex", string)

```
re.findall(r"#movies", "Love #movies! I had fun yesterday going to the #movies")
```

```
['#movies', '#movies']
```

10. The re module
Python has a useful library, the re module, to handle regex. You can import it as shown in the code. Let's see how it works. To find all matches of a pattern, we use the dot findall method. It takes two arguments: the regex and the string. In the code, we want to find all the matches of hashtag movies in the specified string. The method returns a list with the two matches found.

The re module

```
import re
```

- Split string at each match:

re.split(r"regex", string)

```
re.split(r"!", "Nice Place to eat! I'll come back! Excellent meat!")
```

```
['Nice Place to eat', " I'll come back", ' Excellent meat', '']
```

11. The re module
To split a string at each pattern match, we could use the method dot split In the example, we want to split the specified string at every exclamation mark match. It returns a list of the substrings as you can see in the output.

The re module

12. The re module
Finally, we could replace any pattern match with another string using the dot sub method. It takes three arguments: the regex, replacement and string. In the example, we replace every match of yellow with the word nice. We get the following output.

```python
import re
```

- Replace one or many matches with a string:

re.sub(r"regex", new, string)

```python
re.sub(r"yellow", "nice", "I have a yellow car and a yellow house in a yellow neighborhood")
```

```
'I have a nice car and a nice house in a nice neighborhood'
```

Supported metacharacters

Metacharacter	Meaning
\d	Digit

```
re.findall(r"User\d", "The winners are: User9, UserN, User8")
```

```
['User9', 'User8']
```

Metacharacter	Meaning
\D	Non-digit

```
re.findall(r"User\D", "The winners are: User9, UserN, User8")
```

```
['UserN']
```

13. Supported metacharacters
Let's look at the supported metacharacters. In the example, we want to find all matches of the patterns containing User followed by a number. We use backlash d to represent the digit. We get the following matches. Next, we find matches of the pattern containing User followed by a non-digit. In that case, we use backslash capital D obtaining the following match.

Supported metacharacters

Metacharacter	Meaning
\w	Word

```
re.findall(r"User\w", "The winners are: User9, UserN, User8")
```

```
['User9', 'UserN', 'User8']
```

Metacharacter	Meaning
\W	Non-word

```
re.findall(r"\W\d", "This skirt is on sale, only $5 today!")
```

```
['$5']
```

14. Supported metacharacters
If we want to find all matches of the pattern containing User followed by any digit or normal character, we can use backlash w. We get all following matches. In the next example, we need to find the price in a string. We use backslash capital W to match the dollar sign followed by a digit obtaining the following output.

Supported metacharacters

Metacharacter	Meaning
\s	Whitespace

```
re.findall(r"Data\sScience", "I enjoy learning Data Science")
```

```
['Data Science']
```

Metacharacter	Meaning
\S	Non-Whitespace

```
re.sub(r"ice\Scream", "ice cream", "I really like ice-cream")
```

```
'I really like ice cream'
```

15. Supported metacharacters
Finally, we use backslash s to specify the pattern Data whitespace science getting the following match. In the second example, we use backslash capital S to detect the matches of ice, followed by any non-space character, followed by cream and replace them with the word ice cream.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Repetitions

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Science

Repeated characters

Validate the following string:

password1234

5. Repeated characters
To search for a pattern, we can use the method dot search. It takes the regex and string. It tells us if there is a match. Let's apply what we have learned until now. We use backslash w eight times to match the first part and backslash d four times to match the last part. So our method finds a match.

Repeated characters

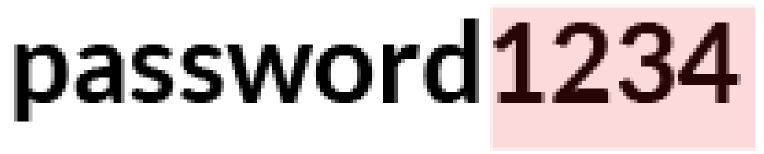
Validate the following string:

password1234

6. Repeated characters
But this seems cumbersome for longer regex. Instead, we can use quantifiers to save this situation. A quantifier is a metacharacter that specifies how many times a character located to its left needs to be matched. In our example, we specify that backslash w is repeated 8 times and backslash d four times. And we get a match as seen in the output.

Repeated characters

Validate the following string:



Repeated characters

Validate the following string:

password1234

```python
import re
password = "password1234"
```

```python
re.search(r"\w\w\w\w\w\w\w\w\d\d\d\d", password)
```

```
<_sre.SRE_Match object; span=(0, 12), match='password1234'>
```

6. Repeated characters
But this seems cumbersome for longer regex. Instead, we can use quantifiers to save this situation. A quantifier is a metacharacter that specifies how many times a character located to its left needs to be matched. In our example, we specify that backslash w is repeated 8 times and backslash d four times. And we get a match as seen in the output.

Repeated characters

Validate the following string:

password1234

```
import re
password = "password1234"
```

```
re.search(r"\w{8}\d{4}", password)
```

```
<_sre.SRE_Match object; span=(0, 12), match='password1234'>
```

Quantifiers:

A metacharacter that tells the regex engine how many times to match a character immediately to its left.

Quantifiers

- Once or more: `+`

```
text = "Date of start: 4-3. Date of registration: 10-04."
```

```
re.findall(r"        ", text)
```

8. Quantifiers
We indicate that the digit, backslash d, should appear once or more times adding the plus quantifier. Then, a dash.

7. Quantifiers
In the following string, we want to match the dates. We can see that the pattern is one or multiple digits, dash, and again one or multiple digits. The plus metacharacter indicates a character that appears once or more times. Let's construct the regex from simple to complex. We will use dot findall method as you can see in the code.

Quantifiers

- Once or more: +

```
text = "Date of start: 4-3. Date of registration: 10-04."
```

```
re.findall(r"\d+-    ", text)
```

Quantifiers

- Once or more: +

```
text = "Date of start: 4-3. Date of registration: 10-04."
```

```
re.findall(r"\d+-\d+", text)
```

```
['4-3', '10-04']
```

9. Quantifiers
And again a digit, backslash d. It should appear once or more times, so again we use the plus quantifier. We get the two matches that we expected as seen in the output.

Quantifiers

- Zero times or more: `*`

```python
my_string = "The concert was amazing! @ameli!a @joh&&n @mary90"
re.findall(r"@\w+\W*\w+", my_string)
```

```
['@ameli!a', '@joh&&n', '@mary90']
```

10. Quantifiers
To indicate that a character should appear zero or more times, we can use the star metacharacter. In this example, we have the following string. We want to find all mentions of users, which start with an @. We notice that they could contain or not contain a non-word character in the middle. So, we construct our regex as seen in the code: an @, followed by backslash w plus, backslash capital w with start to indicate a non-word character zero or more times, then backslash w plus. And we get the matches seen in the output.

Quantifiers

- Zero times or once: ?

```
text = "The color of this image is amazing. However, the colour blue could be brighter."
re.findall(r"colou?r", text)
```

```
['color', 'colour']
```

11. Quantifiers

Another helpful quantifier is the question mark. It indicates that a character should appear zero times or once. In the example, we need to find matches for the word color, which has spelling variations. So, our regex is c, o, l, o, u which can appear once or zero times, so we add the question mark after it, then r. With this regex, we get the two matches wanted.

If a date ends with the year you can use the {n} quantifier to specify that the number is repeated exactly n times.

Quantifiers

- n times at least, m times at most : `{n, m}`

```
phone_number = "John: 1-966-847-3131 Michelle: 54-908-42-42424"
```

```
re.findall(r"                                    ", phone_number)
```

12. Quantifiers
Finally, we can use the curly braces to indicate a specific minimum and maximum times. In the example, we want to find all matches for a phone number. We'll use the dot findall method and construct our regex step by step.

Quantifiers

- n times at least, m times at most : `{n, m}`

```
phone_number = "John: 1-966-847-3131 Michelle: 54-908-42-42424"
```

```
re.findall(r"\d{1,2}-                    ", phone_number)
```

13. Quantifiers
As we can see, we can have a digit, once or twice, then slash.

14. Quantifiers
Then, again a digit, three times. Then, a slash.

Quantifiers

- n times at least, m times at most : `{n, m}`

```
phone_number = "John: 1-966-847-3131 Michelle: 54-908-42-42424"
```

```
re.findall(r"\d{1,2}-\d{3}-           ", phone_number)
```

15. Quantifiers
Then, a digit twice or three times. Then, slash. And finally a digit again. We indicate that this last digit need to appear at least four times leaving the second argument in black. The regex engine returns two matches as expected.

Quantifiers

- n times at least, m times at most : `{n, m}`

```
phone_number = "John: 1-966-847-3131 Michelle: 54-908-42-42424"
```

```
re.findall(r"\d{1,2}-\d{3}-\d{2,3}-\d{4,}", phone_number)
```

```
['1-966-847-3131', '54-908-42-42424']
```

16. Quantifiers
It's very important to remember that the quantifier applies only to the character immediately to the left. As an example, in the regex apple plus, the plus applies only to the letter e and not to the entire word.

Quantifiers

- *Immediately to the left*
  - `r"apple+"` : `+` applies to e and not to apple

16. Quantifiers
It's very important to remember that the quantifier applies only to the character immediately to the left. As an example, in the regex apple plus, the plus applies only to the letter e and not to the entire word.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Regex metacharacters

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

Looking for patterns

Two different operations to find a match:

re.search(r"regex", string)

```
re.search(r"\d{4}", "4506 people attend the show")
```

```
<_sre.SRE_Match object; span=(0, 4), match='4506'>
```

```
re.search(r"\d+", "Yesterday, I saw 3 shows")
```

```
<_sre.SRE_Match object; span=(17, 18), match='3'>
```

re.match(r"regex", string)

```
re.match(r"\d{4}", "4506 people attend the show")
```

```
<_sre.SRE_Match object; span=(0, 4), match='4506'>
```

```
re.match(r"\d+","Yesterday, I saw 3 shows")
```

```
None
```

2. Looking for patterns
Let's first look at two methods of the re module: dot search and dot match. As you can see, they have the same syntax and are used to find a match.. Both methods return an object with the match found. The difference is that dot match is anchored at the **beginning of the string.** In this case, dot search finds a match, but dot match does not. This is because the first characters do not match the regex.

Special characters

- Match any character (except newline): `.`

www.domain.com

```
my_links = "Just check out this link: www.amazingpics.com. It has amazing photos!"
re.findall(r"www  com", my_links)
```

3. Special characters
The dot metacharacter matches any character. In the example code, we need to match links in the string. We know a link starts with w, w, w and ends with c, o, m. So we first write this in our regex.

4. Special characters
We don't know how many character are in between. So we indicate that we want any character, a dot, once or more times, adding the plus. We can see in the output that we get our match.

Special characters

- Match any character (except newline): `.`

www.domain.com

```
my_links = "Just check out this link: www.amazingpics.com. It has amazing photos!"
re.findall(r"www.+com", my_links)
```

```
['www.amazingpics.com']
```

The dot . metacharacter matches any character.

Special characters

- Start of the string: `^`

```
my_string = "the 80s music was much better that the 90s"
```

```
re.findall(r"the\s\d+s", my_string)
```

```
['the 80s', 'the 90s']
```

```
re.findall(r"^the\s\d+s", my_string)
```

```
['the 80s']
```

5. Special characters
The circumflex anchors the regex to the start of a string. In the example, we find the pattern starting with t, followed by h, e, whitespace, two digits and ending with s. The method finds the following two matches. Now, we add the anchor metacharacter. **We receive only one match. The one that appears at the beginning of the string.**

Special characters

- End of the string: $\boxed{\$}$

```
my_string = "the 80s music hits were much better that the 90s"
```

```
re.findall(r"the\s\d+s$", my_string)
```

```
['the 90s']
```

6. Special characters

On the contrary, the dollar sign anchors the regex to the **end of the string.**

If we use it in the previous example, we get the match that appears at the end of the string.

Special characters

- Escape special characters: `\`

```
my_string = "I love the music of Mr.Go. However, the sound was too loud."
```

```
print(re.split(r".\s", my_string))
```

```
['', 'lov', 'th', 'musi', 'o', 'Mr.Go', 'However', 'th', 'soun', 'wa', 'to', 'loud.']
```

```
print(re.split(r"\.\s", my_string))
```

```
['I love the music of Mr.Go', 'However, the sound was too loud.']
```

7. Special characters
What if I want to use characters like dollar sign or dot, which also have other meanings? We want to split the string by dot whitespace. We write the following regex. **And we get an output that it is not what we expected.** Why? Because the regex interprets the dot as any character. **To solve this situation, we need to escape the character by adding a backslash in front of the dot.** Now we get the correct output.

OR operator

- Character: `|`

```
my_string = "Elephants are the world's largest land animal! I would love to see an elephant one day"
```

```
re.findall(r"Elephant|elephant", my_string)
```

```
['Elephant', 'elephant']
```

8. OR operator
In the example code, we want to match the word elephant. However, we see that it's written with capital E or lower e. In that case, we use the vertical bar. In this way, we indicate that we want to match one variant OR the other obtaining both elephant-matches.

OR operator

- Set of characters: `[ ]`

```
my_string = "Yesterday I spent my afternoon with my friends: MaryJohn2 Clary3"
```

```
re.findall(r"[a-zA-Z]+\d", my_string)
```

```
['MaryJohn2', 'Clary3']
```

9. OR operator
Square brackets also represent the OR operand. Inside them, we can specify optional characters to match. Look at the example. We want to find a pattern that contains lowercase or uppercase letter followed by a digit. To do so, we can use the square brackets. Inside them, we will use lowercase a dash lowercase z to specify any lowercase letter. Then, uppercase a dash uppercase z to indicate any uppercase letter. Then the plus. Then backslash d. Thus, we get the following matches.

10. OR operator
In the following string, we want to replace the non-word characters by whitespace. We specify optional characters inside the square brackets. The engine searches for one or the other. When it finds a match, it replaces the character by a whitespace, getting the following output.

OR operator

- Set of characters: `[ ]`

```
my_string = "My&name&is#John Smith. I%live$in#London."
```

```
re.sub(r"[#$%&]", " ", my_string)
```

```
'My name is John Smith. I live in London.'
```

OR operand

- Set of characters: `[ ]`
  - `^` transforms the expression to negative

```
my_links = "Bad website: www.99.com. Favorite site: www.hola.com"
re.findall(r"www[^0-9]+com", my_links)
```

```
['www.hola.com']
```

11. OR operand
The circumflex transforms the expression inside square brackets into negative. In the example, we add the circumflex to specify we want the **links that do not contain any number.** And we get the following output.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Greedy vs. non-greedy matching

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

Greedy vs. non-greedy matching

- Two types of matching methods:
  - Greedy
  - Non-greedy or lazy
- Standard quantifiers are greedy by default: `*`, `+`, `?`, `{num, num}`

2. Greedy vs. non-greedy matching
There are two types of matching methods: greedy and non-greedy (also called lazy) operators. The quantifiers that you have been learning until now (which are called standard quantifiers) are greedy by default.

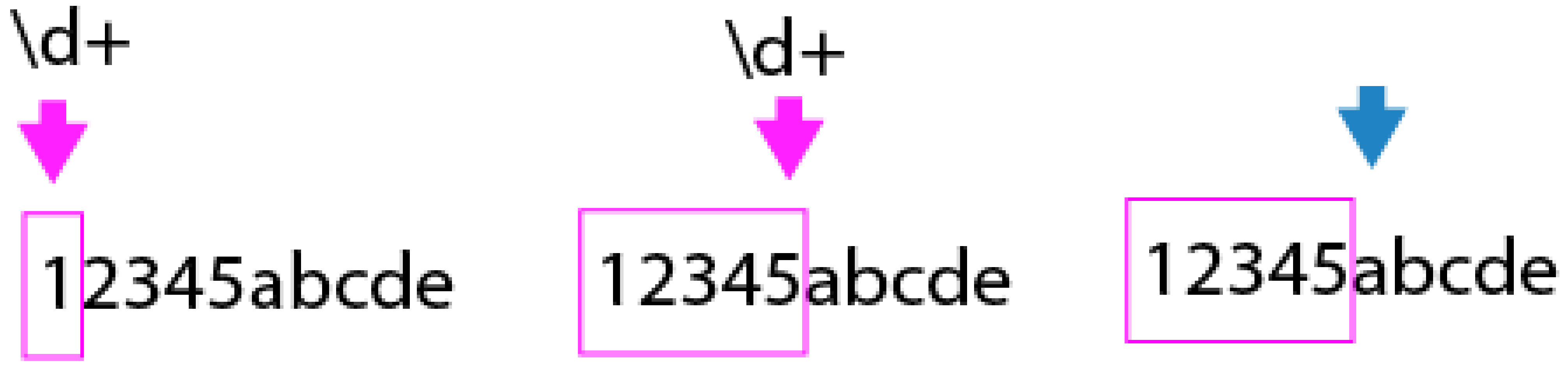
Greedy matching

- **Greedy:** match as many characters as possible
- Return the *longest match*

```
import re
re.match(r"\d+", "12345bcada")
```

```
<_sre.SRE_Match object; span=(0, 5), match='12345'>
```

3. Greedy matching
We said that the standard quantifiers have a greedy behavior, meaning that they will attempt to match as many characters as possible. And in doing so, they will return the longest match found with a match attempt. Let's take a look at this code. We want to find a pattern that has one or more digits on the string displayed and our greedy quantifier will return the pattern '12345'. We can explain this in the following way: our quantifier will start by matching the first digit found, '1'. Because it is greedy, it will keep going to find 'more' digits and stop only when no other digit can be matched, returning '12345'.



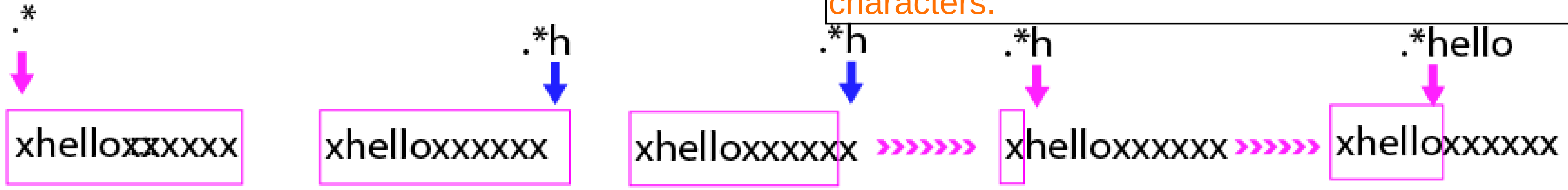
Greedy matching

- Backtracks when too many character matched
- Gives up characters one at a time

```
import re
re.match(r".*hello", "xhelloxxxxxx")
```

```
<_sre.SRE_Match object; span=(0, 6), match='xhello'>
```

4. Greedy matching
However, If the greedy quantifier has matched so many characters that can not match the rest of pattern, it will backtrack, giving up characters matched earlier one at a time and try again. Backtracking is like driving a car without a map. If you drive through a path and hit a dead end street, you need to backtrack along your road to an earlier point to take another street. For eg., We use the greedy quantifier .* to find anything, zero or more times, followed by the letters "h" "e" "l" "l" "o". We can see here that it returns the pattern 'xhello'. So our greedy quantifier will start by matching as much as possible, the entire string. Then it tries to match the h, but there are no characters left. So it backtracks, giving up one matched character. Trying again. It still doesn't match the h, so it backtracks one more step repeatedly till it finally matches the h in the regex, and the rest of the characters.



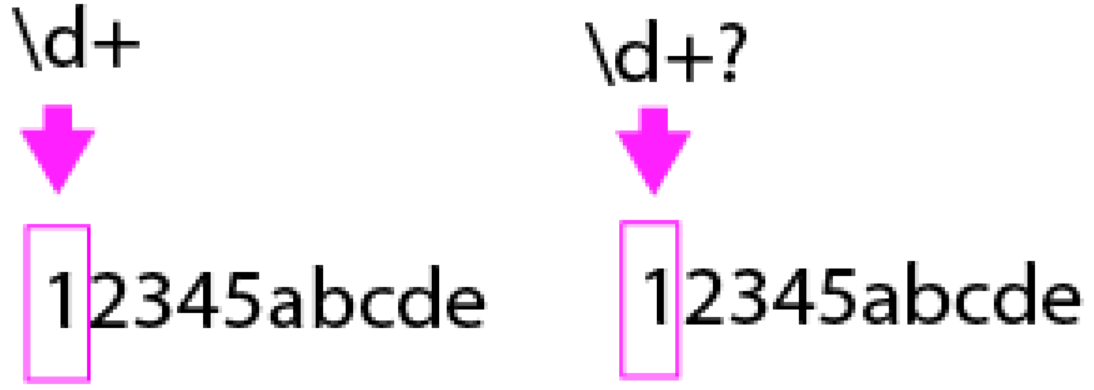
Non-greedy matching

- **Lazy:** match as few characters as needed
- Returns *the shortest match*
- Append `?` to greedy quantifiers

```
import re
re.match(r"\d+?", "12345bcada")
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

5. Non-greedy matching
Because they have lazy behavior, non-greedy quantifiers will attempt to match as few characters as needed returning the shortest match. So how do we obtain non-greedy quantifiers? We can append a question mark at the end of the greedy quantifiers to convert them into lazy. If we take the same code as before, our non-greedy quantifier will return the pattern '1'. In this case, our quantifier will start by matching the first digit found, '1'. Because it is non-greedy, it will stop there as we stated that we want 'one or more' and one is as few as needed.

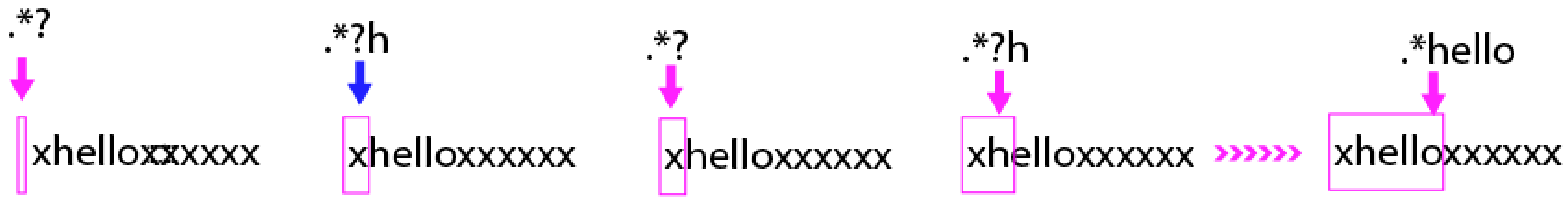


Non-greedy matching

- Backtracks when too few characters matched
- Expands characters one a time

```python
import re
re.match(r".*?hello", "xhelloxxxxxx")
```

```
<_sre.SRE_Match object; span=(0, 6), match='xhello'>
```



6. Non-greedy matching
Non-greedy quantifiers also backtrack. In this case, if they have matched so few characters that the rest of the pattern cannot match, they backtrack, expand the matched character one at a time and try again. Let's take the same example code again. This time we will use the lazy quantifier .*?. Interestingly, we obtain the same match 'xhello'. But how this match was obtained is different from the first time. The lazy quantifier first matches as little as possible, nothing, leaving the entire string unmatched. Then it tries to match the h, but it doesn't work. So it backtracks, matching one more character, the x. Then it tries again, this time matching the h, and afterwards, the rest of the regex.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON