Grouping and capturing

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Group characters

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

Group characters

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

```
re.findall(r'[A-Za-z]+\s\w+\s\d+\s\w+', text)
```

```
['Clary has 2 friends', 'Susan has 3 brothers', 'John has 4 sisters']
```

3. Group characters
And we want to extract information about a person, how many and which type of relationships they have. So, we want to extract Clary 2 friends, Susan 3 brothers and John 4 sisters, as you can see in the slide. We know the structure of the sentences. Let's try our first approach. We would write something like in the code, any upper or lowercase letter, whitespace, any word character, whitespace, a number, whitespace and any word character. Let's see the output. Quite close. But we don't want the word has.

Capturing groups

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

- Use parentheses to **group** and **capture** characters together

```
([A-Za-z]+)\s\w+\s\d+\s\w+
```

Group

4. Capturing groups
What can we do about this? We start simple by trying to extract only the names. We can place parentheses to group those characters as shown in the slide. Capture them. **And retrieve only that group.**

5. Capturing groups
In the code, we have now added parentheses to group our first part of the regex. We can observe in the output that the group was captured. And only the three names were retrieved.

Capturing groups

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

- Use parentheses to group and capture characters together

```
re.findall(r'([A-Za-z]+)\s\w+\s\d+\s\w+', text)
```

```
['Clary', 'Susan', 'John']
```

6. Capturing groups
Let's look at the example again. We can place parentheses around the three groups that we want to capture as shown in the slide. Each group will receive a number. The entire expression will always be group zero. The first group one, the second two, and the third number three. We'll see how to use these numbers later.

Capturing groups

7. Capturing groups
Let's see this in the code example. We add the parentheses to group together each of the three parts of the regex. In the output, we got a list of tuples. The first element of each tuple is the match captured corresponding to group one. The second to group two. The last to group three.

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

Group 0

([A-Za-z]+)\s\w+\s(\d+)\s(\w+)

Group1 Group2 Group3

Capturing groups

8. Capturing groups
As we already discussed, we can use capturing groups to match a specific subpattern in a pattern. We can use this information for retrieving the groups by numbers as we'll learn later in the next videos.

Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

```
re.findall(r'([A-Za-z]+)\s\w+\s(\d+)\s(\w+)', text)
```

```
[('Clary', '2', 'friends'),
 ('Susan', '3', 'brothers'),
 ('John', '4', 'sisters')]
```

Capturing groups

- Match a specific subpattern in a pattern
- Use it for further processing

Capturing groups

- Organize the data

```
pets = re.findall(r'([A-Za-z]+)\s\w+\s(\d+)\s(\w+)', "Clary has 2 dogs but John has 3 cats")
pets[0][0]
```

```
'Clary'
```

9. Capturing groups
But we can also use it to organize data. As you saw earlier, the matches are retrieved as lists. In the code, we placed the parentheses to capture the name of the owner, the number and which type of pets each one has. We can access the information retrieved by using indexing and slicing as seen in the code.

Capturing groups

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

- Apply a quantifier to the entire group

```
re.search(r"(\d[A-Za-z])+", "My user name is 3e4r5fg")
```

```
<_sre.SRE_Match object; span=(16, 22), match='3e4r5f'>
```

10. Capturing groups
But capturing groups have one important feature. Remember that quantifiers apply to the character immediately to the left. So, we can place parentheses to group characters and then apply the quantifier to the entire group. In the code, we have placed parentheses to match the group containing a number and any letter. We applied the plus quantifier to specify that we want this group repeated once or more times. And we get the following match shown in the output.

Capturing groups

- Capture a repeated group `(\d+)` vs. repeat a capturing group `(\d)+`

```
my_string = "My lucky numbers are 8755 and 33"
re.findall(r"(\d)+", my_string)
```

```
['5', '3']
```

11. Capturing groups
But be careful. It's not the same to capture a repeated group than to repeat a capturing group. In the first code, we use findall to match a capturing group containing one number. We want this capturing group to be repeated once or more times. We get 5 and 3 as an output. Because these numbers are repeated consecutively once or more times. In the second code, we specify that we should capture a group containing one or more repetitions of a number.

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

```
['8755', '33']
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Alternation and non-capturing groups

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Pipe

- Vertical bar or pipe: `|`

```
my_string = "I want to have a pet. But I don't know if I want a cat, a dog or a bird."
re.findall(r"cat|dog|bird", my_string)
```

```
['cat', 'dog', 'bird']
```

2. Pipe
And we want to find all matches for pet names. So we can use the pipe operator to specify that we want to match cat or dog or bird as you see in the code.

Pipe

- Vertical bar or pipe: `|`

3. Pipe
Now, we changed the string a little bit. And once more we want to find all the pet names. But this time only those that come after a number and a whitespace. So we specify this again with the pipe operator. Hmm we got the wrong output. Why? The pipe operator works comparing everything that is to its left (digit whitespace cat) with everything to the right, dog.

```
my_string = "I want to have a pet. But I don't know if I want 2 cats, 1 dog or a bird."
re.findall(r"\d+\scat|dog|bird", my_string)
```

```
['2 cat', 'dog', 'bird']
```

Alternation

- Use groups to choose between optional patterns


```
my_string = "I want to have a pet. But I don't know if I want 2 cats, 1 dog or a bird."
re.findall(r"\d+\s(cat|dog|bird)", my_string)
```

```
['cat', 'dog']
```

4. Alternation
In simpler terms, we can use parentheses again to group the optional characters as you can see in the slide. In the code, now the parentheses are added to group cat or dog or bird. This time we get the output cat and dog. This is the correct match as only these two patterns followed a number and whitespace.

Alternation

- Use groups to choose between optional patterns


```
my_string = "I want to have a pet. But I don't know if I want 2 cats, 1 dog or a bird."
re.findall(r"(\d)+\s(cat|dog|bird)", my_string)
```

```
[('2', 'cat'), ('1', 'dog')]
```

5. Alternation
In the previous example, we may also want to match the number. In that case, we need to place parentheses to capture the digit group as seen in the slide. In the code, we now use two pair of parentheses. We use findall in the string. And we get a list with two tuples as shown in the output.

Non-capturing groups

- Match but **not capture** a group
  - When group is not backreferenced
  - Add `?:` : `(?:regex)`

6. Non-capturing groups
Sometimes, we need to group characters using parentheses. But we are not going to reference back to this group. For these cases, there are a special type of groups called non-capturing groups. For using them, we just need to add question mark colon inside the parenthesis **but before the regex.**

Non-capturing groups

- Match but **not capture** a group

(?:\d{2}-){3}(\d{3}-\d{3})

Group 1

```
my_string = "John Smith: 34-34-34-042-980, Rebeca Smith: 10-10-10-434-425"
re.findall(r"(?:\d{2}-){3}(\d{3}-\d{3})", my_string)
```

```
['042-980', '434-425']
```

7. Non-capturing groups
We have the following string. We want to find all matches of numbers. We see that the pattern consists of two numbers and dash repeated three times. After that, three numbers, dash, four numbers. We want to extract only the last part without the first repeated elements. We need to group the first two elements to indicate repetitions. But we don't want to capture them. So we use non-capturing groups to group backslash d repeated two times and dash. Then we indicate this group should be repeated three times. Then, we group backslash d repeated three times, dash, backslash d repeated three times as shown in the slide. In the code, we then match the regex to the string. And we get the numbers we were looking for as shown in the output.

Alternation

- Use non-capturing groups for alternation

```
my_date = "Today is 23rd May 2019. Tomorrow is 24th May 19."
re.findall(r"(\d+)(?:th|rd)", my_date)
```

```
['23', '24']
```

8. Alternation
Finally, we can combine non-capturing groups and alternation together. Remember that alternation implies using parentheses and the pipe operand to group optional characters. Let's suppose that we have the following string. And we want to match all the numbers of the day. We know that they are followed by th or rd. But we only want to capture the number and not the letters that follow...

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Backreferences

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Numbered groups

Python 3.0 was released on 12-03-2008. It was a major revision of the language. Many of its major features were backported to Python 2.6.x and 2.7.x version series.

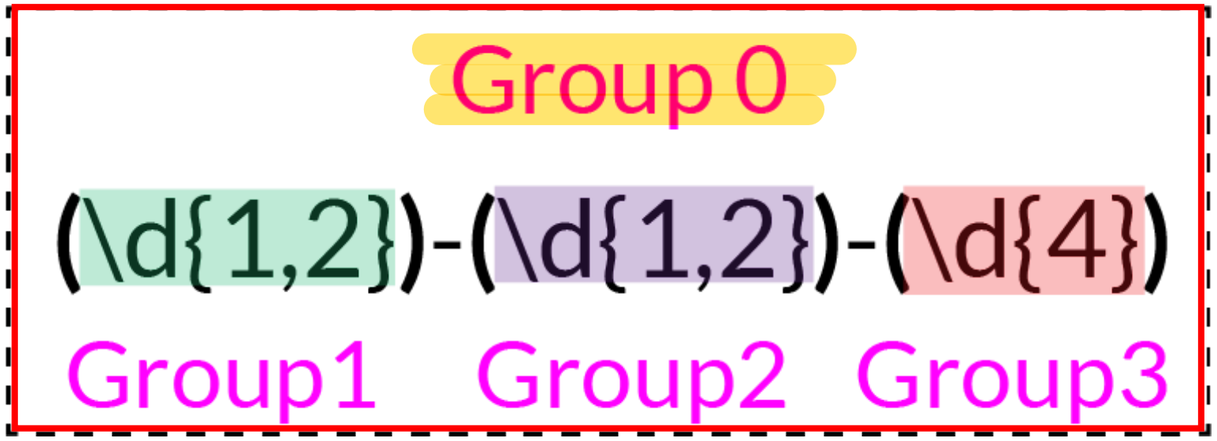
(\d{1,2})-(\d{1,2})-(\d{4})

2. Numbered groups
Imagine we come across this text. And we want to extract the date highlighted. But we want to extract only the numbers. So, we can place parentheses in a regex to capture these groups as we learned.

Numbered groups

3. Numbered groups
We have also seen that each of these groups receive a number. The whole expression is group zero. The first group one, and so on, as shown in the slide.

Python 3.0 was released on 12-03-2008. It was a major revision of the language. Many of its major features were backported to Python 2.6.x and 2.7.x version series.



Numbered groups

4. Numbered groups
Let's use dot search to match the pattern to the text. To retrieve the groups captured, we can use the method dot group specifying the number of a group we want. For example, three. The method retrieves the match corresponding to group number three as shown in the output. We can also retrieve group zero. It will output the entire expression. Dot group can only be used with dot search and dot match methods.

```
text = "Python 3.0 was released on 12-03-2008."
```

```
information = re.search('(\d{1,2})-(\d{2})-(\d{4})', text)
information.group(3)
```

```
'2008'
```

```
information.group(0)
```

```
'12-03-2008'
```

Named groups

- Give a name to groups

5. Named groups
We can also give names to our capturing groups. Inside the parentheses, we write question mark capital p, and the name inside angle brackets as shown in the slide.

Named groups

- Give a name to groups

```
text = "Austin, 78701"
cities = re.search(r"(?P<city>[A-Za-z]+).*?(?P<zipcode>\d{5})", text)
cities.group("city")
```

```
'Austin'
```

```
cities.group("zipcode")
```

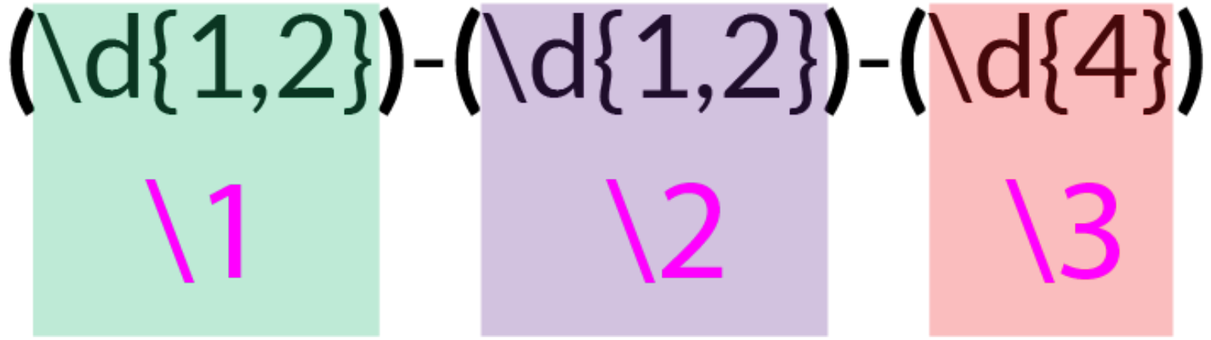
```
'78701'
```

6. Named groups

Let's say we have the following string. We want to match the name of the city and zipcode in different groups. We can use capturing groups and assign them the name city and zipcode as shown in the code. We retrieve the information by using dot group. We indicate the name of the group. For example, specifying city gives us the output Austin. Specifying zipcode gives us the number match as shown.

Backreferences

- Using capturing groups to reference back to a group



7. Backreferences
There is another way to backreference groups. In fact, the matched group can be reused inside the same regex or outside for substitution. We can do this using backslash and the number of the group as you can see in the slide.

8. Backreferences
Let's see an example. We have the following string. We want to find all matches of repeated words. In the code, we specify that we want to capture a sequence of word characters. Then a whitespace.

Backreferences

- Using numbered capturing groups to reference back

```
sentence = "I wish you a happy happy birthday!"
re.findall(r"(\w+)\s  ", sentence)
```

9. Backreferences
Then we write backslash one. This will indicate that we want to match the first group captured again. In other words, it says match that sequence of characters that was previously captured once more. And we get the word happy as an output. This was the repeated word in our string.

Backreferences

- Using numbered capturing groups to reference back

```
sentence = "I wish you a happy happy birthday!"
re.findall(r"(\w+)\s\1", sentence)
```

```
['happy']
```

10. Backreferences
Now, we will replace the repeated word with one occurrence of the same word. In the code, we use the same regex as before. This time, we use the dot sub method. In the replacement part, we can also reference back to the captured group. We write r backslash one inside quotes. This says replace the entire expression match with the first captured group. In the output string, we have only one occurrence of the word happy.

Backreferences

- Using numbered capturing groups to reference back

```
sentence = "I wish you a happy happy birthday!"
re.sub(r"(\w+)\s\1", r"\1", sentence)
```

```
'I wish you a happy birthday!'
```

Backreferences

- Using named capturing groups to reference back

(?P<name>regex)

?P=name

```
sentence = "Your new code number is 23434. Please, enter 23434 to open the door."
re.findall(r"(?P<code>\d{5}).*?(?P=code)", sentence)
```

```
['23434']
```

11. Backreferences
We can also use named groups for backreferencing. To do this, we use question mark capital p equal sign and the group name. In the code, we want to find all matches of the same number. We use a capturing group and name it code. Later, we reference back to this group. And we obtain the number as an output.

Backreferences

- Using named capturing groups to reference back

(?P<name>regex)

\g<name>

12. Backreferences
On the other hand, to reference the group back for replacement we need to use backslash g and the group name inside angle brackets. In the code, we want to replace repeated words by one occurrence of the same word. Inside the regex, we use the previous syntax. In the replacement field, we need to use this new syntax as seen in the code to get the following output.

```
sentence = "This app is not working! It's repeating the last word word."
re.sub(r"(?P<word>\w+)\s(?P=word)", r"\g<word>", sentence)
```

```
'This app is not working! It's repeating the last word.'
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Lookaround

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Looking around

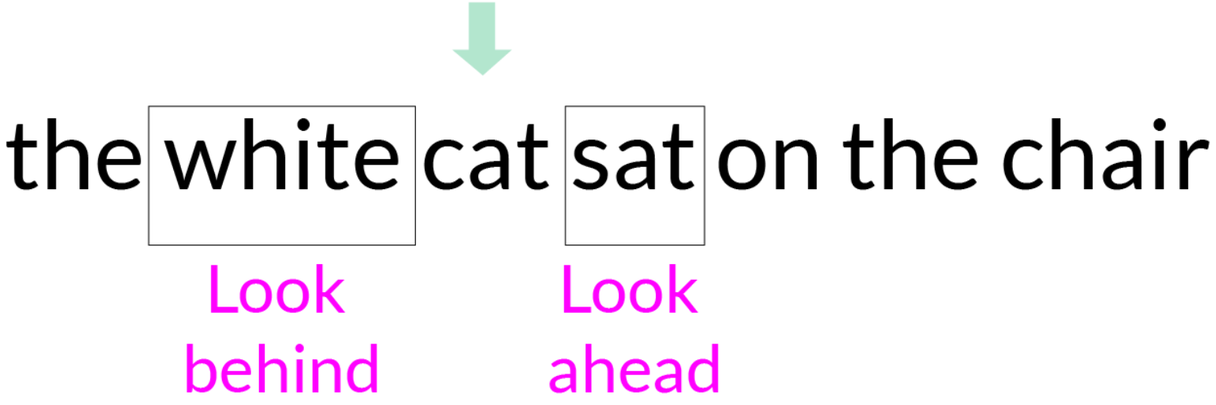
- Allow us to confirm that sub-pattern is ahead or behind main pattern

the white cat sat on the chair

2. Looking around
Look-around will look for what is behind or ahead of a pattern. Imagine that we have the following string.
3. Looking around
We want to see what is surrounding a specific word. For example, we position ourselves in the word cat. So look-around will let us answer the following problem. At my current position, look ahead and search if sat is there. Or look behind and search if white is there.

Looking around

- Allow us to confirm that sub-pattern is ahead or behind main pattern



> At my current position in the matching process, look ahead or behind and examine whether some pattern matches or not match before continuing.

4. Look-ahead
We'll start by exploring look-ahead. This non-capturing group checks whether the first part of the expression is followed or not by the lookahead expression. As a consequence, it will return the first part of the expression. In the previous example, we are looking for the word cat. The look ahead expression can be either positive or negative. For positive we use question mark equal. For negative question mark exclamation mark.

Look-ahead

- Non-capturing group
- Checks that the first part of the expression is followed or not by the lookahead expression
- Return only the first part of the expression

the white cat sat on the chair

Look ahead

positive (?=sat) negative (?!run)

4. Look-ahead
We'll start by exploring look-ahead. This non-capturing group checks whether the first part of the expression is followed or not by the lookahead expression. As a consequence, it will return the first part of the expression. In the previous example, we are looking for the word cat. The look ahead expression can be either positive or negative. For positive we use question mark equal. For negative question mark exclamation mark.

Positive look-ahead

- Non-capturing group
- Checks that the first part of the expression is followed by the lookahead expression
- Return only the first part of the expression

```
my_text = "tweets.txt transferred, mypass.txt transferred, keywords.txt error"
re.findall(r"\w+\.txt                    ", my_text)
```

5. Positive look-ahead
Let's start with positive lookahead. Let's imagine that we have a string containing file names and the status of that file as shown in the code. We want to extract only those files that are followed by the word transferred. So we start building the regex by indicating any word character followed by dot txt.

Positive look-ahead

- Non-capturing group
- Checks that the first part of the expression is followed by the lookahead expression
- Return only the first part of the expression

```
my_text = "tweets.txt transferred, mypass.txt transferred, keywords.txt error"
re.findall(r"\w+\.txt(?=\stransferred)", my_text)
```

```
['tweets.txt', 'mypass.txt']
```

6. Positive look-ahead
We now indicate we want the first part to be followed by the word transferred. We do so by writing question mark equal and then whitespace transferred all inside the parenthesis. With that specification, we get only the desired strings as shown in the output.

Negative look-ahead

- Non-capturing group
- Checks that the first part of the expression is **not** followed by the lookahead expression
- Return only the first part of the expression

```
my_text = "tweets.txt transferred, mypass.txt transferred, keywords.txt error"
re.findall(r"\w+\.txt                    ", my_text)
```

7. Negative look-ahead
Now, let's use negative lookahead in the same example.

8. Negative look-ahead
In this case, we will say that we want those matches that are NOT followed by the expression transferred. We use instead question mark exclamation mark inside parenthesis as seen in the code. Now, we get this other output.

Negative look-ahead

- Non-capturing group
- Checks that the first part of the expression is **not** followed by the lookahead expression
- Return only the first part of the expression

```
my_text = "tweets.txt transferred, mypass.txt transferred, keywords.txt error"
re.findall(r"\w+\.txt(?!\stransferred)", my_text)
```

```
['keywords.txt']
```

8. Negative look-ahead
In this case, we will say that we want those matches that are NOT followed by the expression transferred. We use instead question mark exclamation mark inside parenthesis as seen in the code. Now, we get this other output.

Look-behind

- Non-capturing group
- Get all the matches that are preceded or not by a specific pattern.
- Return pattern after look-behind expression

the white cat sat on the chair.

Look behind

positive (?<=white)

negative (?<!brown)

9. Look-behind
The non-capturing group look-behind gets all matches that are preceded or not by a specific pattern. As a consequence, it will return the matches after the look expression. Let's use the same example, but now we are looking before the word cat. Look behind expression can also be either positive or negative. For positive we use question mark angle bracket equal. For negative question mark angle bracket exclamation mark.

Positive look-behind

- Non-capturing group
- Get all the matches that are preceded by a specific pattern.
- Return pattern after look-behind expression

```
my_text = "Member: Angus Young, Member: Chris Slade, Past: Malcolm Young, Past: Cliff Williams."
re.findall(r"               \w+\s\w+", my_text)
```

10. Positive look-behind
Let's look at the following string. We want to find all matches of the names that are preceded by the word member. How do we construct our regex with positive look-behind? Let's examine the code. At the end of the regex, we'll indicate we want a sequence of word characters whitespace another sequence of word characters.

Positive look-behind

- Non-capturing group
- Get all the matches that are preceded by a specific pattern.
- Return pattern after look-behind expression

```
my_text = "Member: Angus Young, Member: Chris Slade, Past: Malcolm Young, Past: Cliff Williams."
re.findall(r"(?<=Member:\s)\w+\s\w+", my_text)
```

```
['Angus Young', 'Chris Slade']
```

11. Positive look-behind
Pay attention to the code. The look-behind expression goes before that expression. We indicate question mark angle bracket equal followed by member, colon, and whitespace. All inside parentheses. In that way we get the two names that were preceded by the word member as shown in the output.

Negative look-behind

- Non-capturing group
- Get all the matches that are **not** preceded by a specific pattern.
- Return pattern after look-behind expression

```
my_text = "My white cat sat at the table. However, my brown dog was lying on the couch."
re.findall(r"(?<!brown\s)(cat|dog)", my_text)
```

```
['cat']
```

12. Negative look-behind
Now, we have this other string. We will use negative look-behind. We will find all matches of the word cat or dog that are not preceded by the word brown. In this code example, we use question mark angle bracket exclamation mark, followed by brown, whitespace. All inside the parenthesis. Then, we indicate our alternation group: cat or dog. Consequently, we get cat as an output. The cat or dog word that is not after the word brown.

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Finishing line

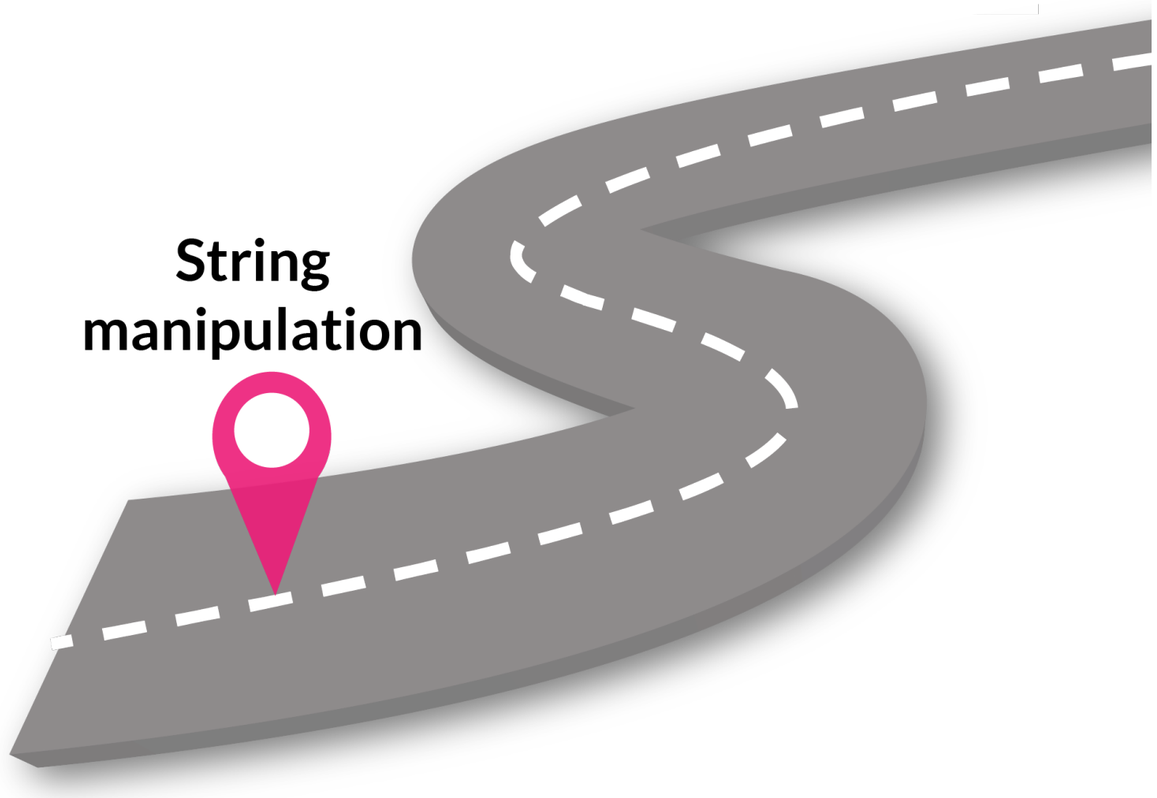
REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

```
r"(Congratulations!)+"
```

Our journey



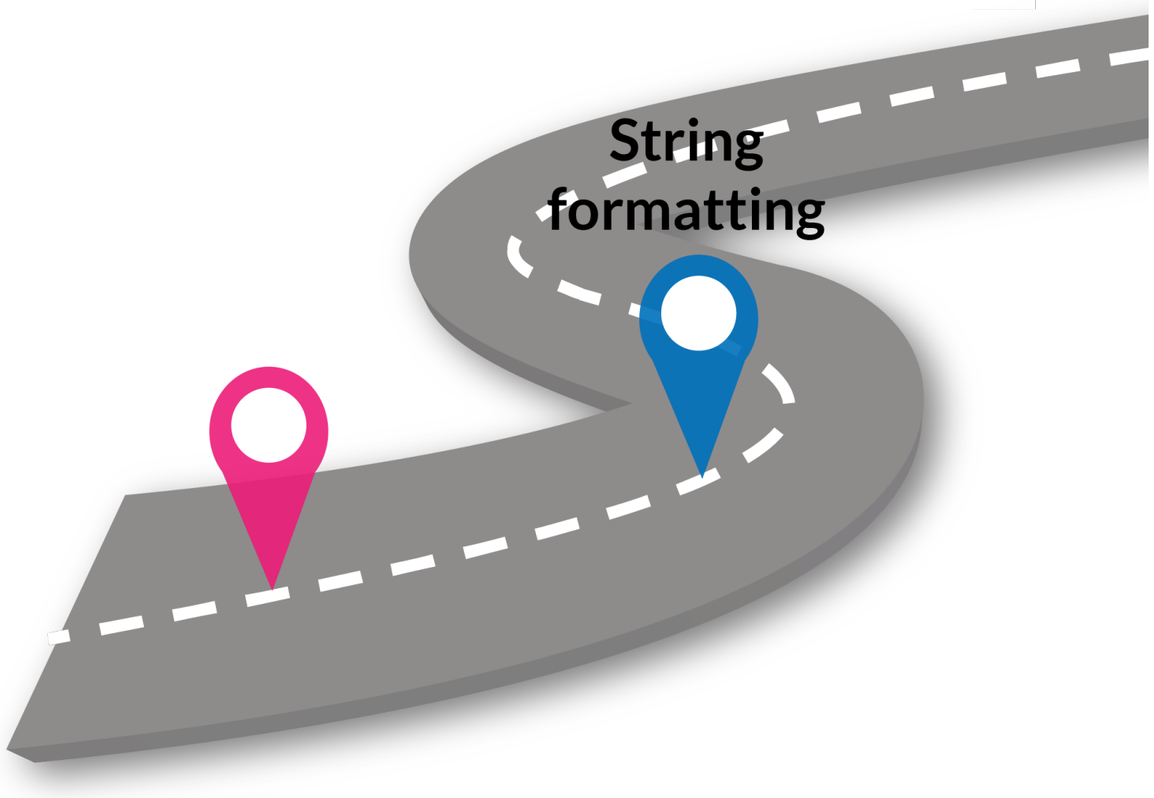
Key concepts

Concatenate and split

Index and slice strings

Replace and remove characters

Our journey



Insert custom strings into a predefined text

Three string formatting methods

Best approach according to situation

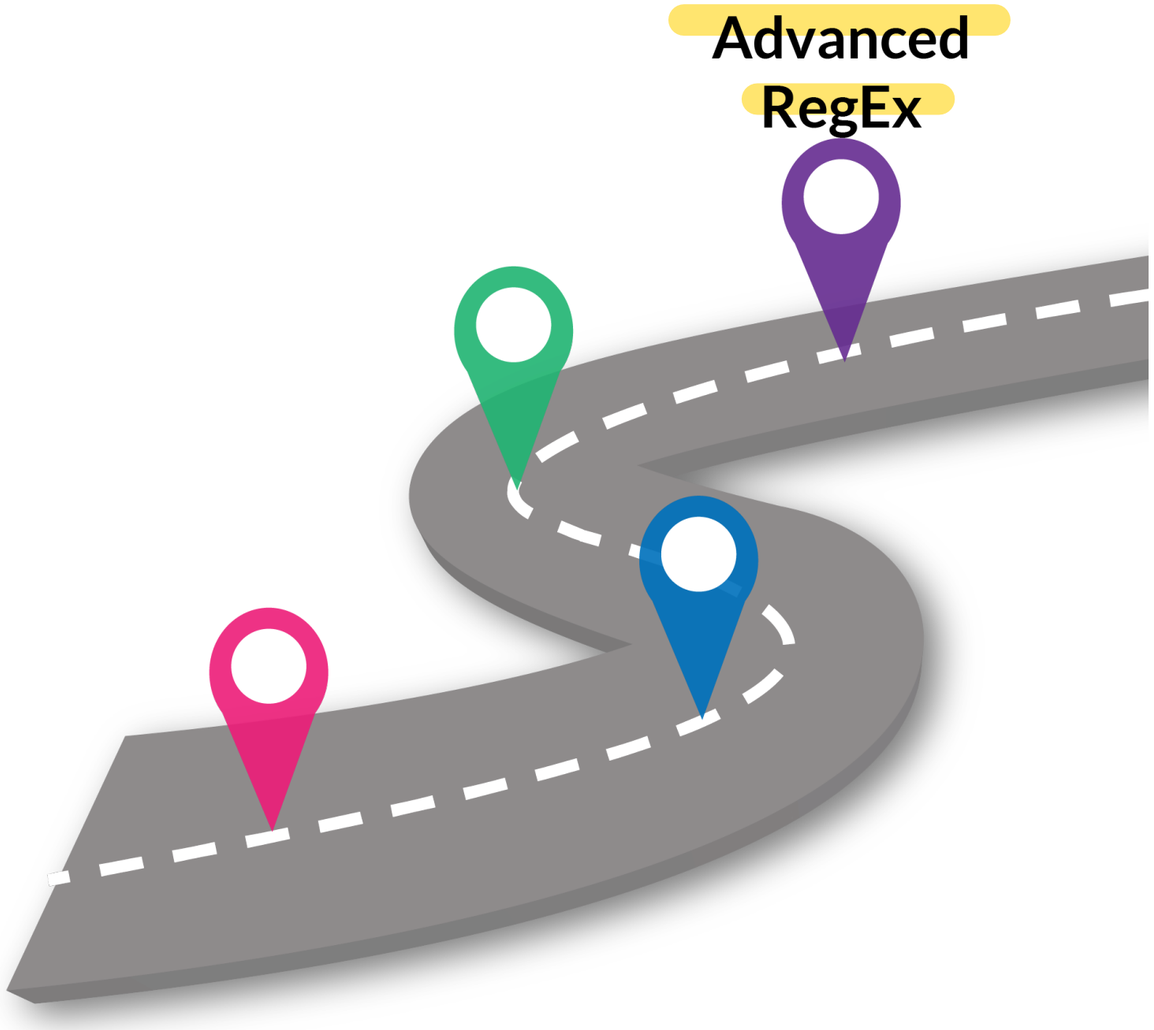
Our journey

Basic concepts

RegEx

- Basic syntax
- Normal characters
- Metacharacters
- Greedy and non-greedy quantifiers

Our journey



Capturing and non-capturing groups

Backreference a pattern

Lookaround an expression

Last tips

- ✓ Practice
- ✓ Apply
- ✓ Have fun

Thank you!

REGULAR EXPRESSIONS IN PYTHON