Introduction to string manipulation

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

You will learn

- **String manipulation**
  - e.g. replace and find specific substrings
- **String formatting**
  - e.g. interpolating a string in a template
- **Basic and advanced regular expressions**
  - e.g. finding complex patterns in a string

Why it is important

- Clean dataset to prepare it for text mining or sentiment analysis
- Process email content to feed a machine learning algorithm that decides whether an email is spam
- Parse and extract specific data from a website to build a database

Strings

- Sequence of characters
- Quotes

```
my_string = "This is a string"
my_string2 = 'This is also a string'
```

```
my_string = 'And this? It's the wrong string'
```

```
my_string = "And this? It's the correct string"
```

More strings

- Length

```
my_string = "Awesome day"
len(my_string)
```

```
11
```

- Convert to string

```
str(123)
```

```
'123'
```

Concatenation

- Concatenate: `+` operator

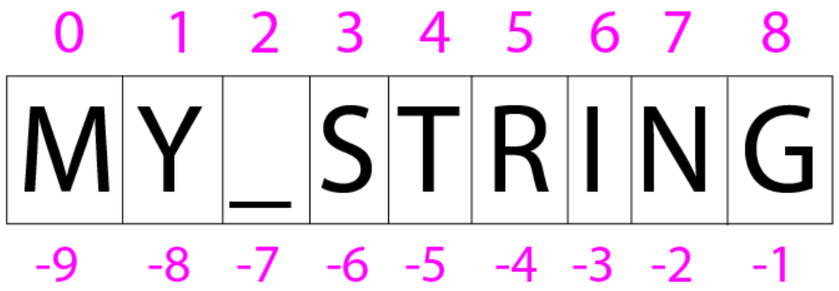
```
my_string1 = "Awesome day"
my_string2 = "for biking"
```

```
print(my_string1+" "+my_string2)
```

```
Awesome day for biking
```

Indexing

- **Bracket notation**




```
my_string = "Awesome day"
```

```
print(my_string[3])
```

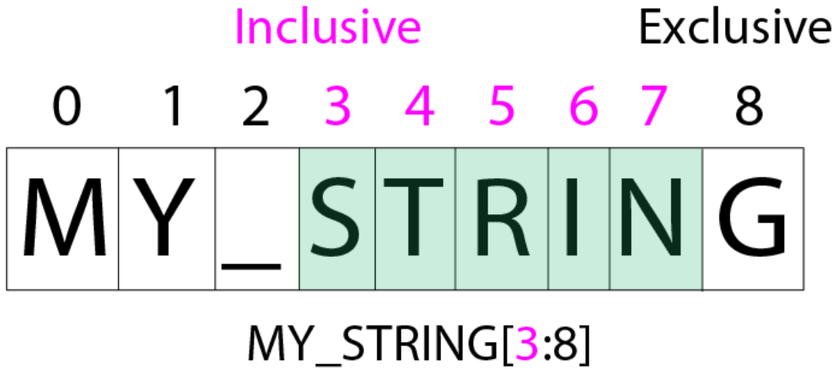
```
s
```

```
print(my_string[-1])
```

```
y
```

Slicing

- Bracket notation




```
my_string = "Awesome day"
print(my_string[0:3])
```

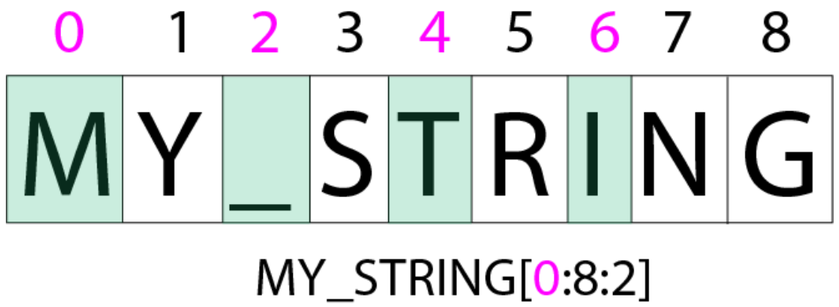
```
Awe
```

```
print(my_string[:5])
print(my_string[5:])
```

```
Aweso
me day
```

Stride

- Specifying stride




```
my_string = "Awesome day"
print(my_string[0:6:2])
```

```
Aeo
```

```
print(my_string[::-1])
```

```
yad emosewA
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

String operations

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Adjusting cases

```
my_string = "tHis Is a niCe StriNg"
```

- Converting to lowercase

```
print(my_string.lower())
```

```
this is a nice string
```

- Converting to uppercase

```
print(my_string.upper())
```

```
THIS IS A NICE STRING
```

```
my_string = "tHis Is a niCe StriNg"
```

- Capitalizing the first character

```
print(my_string.capitalize())
```

```
This is a nice string
```

Splitting

```
my_string = "This string will be split"
```

- Splitting a string into a list of substrings

```
my_string.split(sep=" ", maxsplit=2)
```

```
['This', 'string', 'will be split']
```

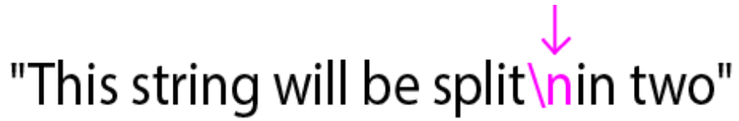
```
my_string.rsplit(sep=" ", maxsplit=2)
```

```
['This string will', 'be', 'split']
```

```
my_string = "This string will be split\nin two"
print(my_string)
```

```
This string will be split
in two
```

Escape Sequence	Character
\n	Newline
\r	Carriage return



- Breaking at line boundaries

```
my_string = "This string will be split\nin two"
```

```
my_string.splitlines()
```

```
['This string will be split', 'in two']
```

Joining

- Concatenate strings from list or another iterable

sep.join(iterable)

```
my_list = ["this", "would", "be", "a", "string"]
print(" ".join(my_list))
```

```
this would be a string
```

Stripping characters

- Strips characters from left to right: `.strip()`

```
my_string = " This string will be stripped\n"
```

```
my_string.strip()
```

```
'This string will be stripped'
```

```
my_string = " This string will be stripped\n"
```

- Remove characters from the right end

```
my_string.rstrip()
```

```
' This string will be stripped'
```

- Remove characters from the left end

```
my_string.lstrip()
```

```
'This string will be stripped\n'
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Finding and replacing

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data scientist

datacamp

Finding substrings

- Search target string for a specified substring.

string.find(substring, start, end)

optional

```
my_string = "Where's Waldo?"
my_string.find("Waldo")
```

```
8
```

```
my_string.find("Wenda")
```

```
-1
```

Finding substrings

- Search target string for a specified substring.


```
my_string = "Where's Waldo?"
```

```
my_string.find("Waldo", 0, 6)
```

```
-1
```

Index function

- Similar to `.find()`, search target string for a specified substring.


```
my_string = "Where's Waldo?"
my_string.index("Waldo")
```

```
8
```

```
my_string.index("Wenda")
```

```
File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Index function

- Similar to `.find()`, search target string for a specified substring.


```
my_string = "Where's Waldo?"
```

```
try:
    my_string.index("Wenda")
except ValueError:
      print("Not found")
```

```
"Not found"
```

Counting occurrences

- Return number of occurrences for a specified substring.

string.count(substring, start, end)

optional

```
my_string = "How many fruits do you have in your fruit basket?"
my_string.count("fruit")
```

```
2
```

```
my_string.count("fruit", 0, 16)
```

```
1
```

Replacing substrings

- Replace occurrences of substring with new substring.


```
my_string = "The red house is between the blue house and the old house"
print(my_string.replace("house", "car"))
```

```
The red car is between the blue car and the old car
```

```
print(my_string.replace("house", "car", 2))
```

```
The red car is between the blue car and the old house
```

Wrapping up

- **String manipulation:**
  - Slice and concatenate
  - Adjust cases
  - Split and join
  - Remove characters from beginning and end
  - Finding substrings
  - Counting occurrences
  - Replacing substrings

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Positional formatting

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data scientist

datacamp

What is string formatting?

- *String interpolation*
- Insert a custom string or variable in predefined text:

```
custom_string = "String formatting"
print(f"{custom_string} is a powerful technique")
```

```
String formatting is a powerful technique
```

- Usage:
  - Title in a graph
  - Show message or error
  - Pass statement a to function

Methods for formatting

- Positional formatting
- Formatted string literals
- Template method

Positional formatting

- Placeholder replace by value

- `str.format()`

```
print("Machine learning provides {} the ability to learn {}".format("systems", "automatically"))
```

```
Machine learning provides systems the ability to learn automatically
```

Positional formatting

- Use variables for both the initial string and the values passed into the method

```
my_string = "{} rely on {} datasets"
method = "Supervised algorithms"
condition = "labeled"
```

```
print(my_string.format(method, condition))
```

```
Supervised algorithms rely on labeled datasets
```

Reordering values

- Include an index number into the placeholders to reorder values

```
print("{} has a friend called {} and a sister called {}".format("Betty", "Linda", "Daisy"))
```

```
Betty has a friend called Linda and a sister called Daisy
```

```
print("{2} has a friend called {0} and a sister called {1}".format("Betty", "Linda", "Daisy"))
```

```
Daisy has a friend called Betty and a sister called Linda
```

Named placeholders

- Specify a name for the placeholders

```
tool="Unsupervised algorithms"
goal="patterns"
print("{title} try to find {aim} in the dataset".format(title=tool, aim=goal))
```

```
Unsupervised algorithms try to find patterns in the dataset
```

Named placeholders

```
my_methods = {"tool": "Unsupervised algorithms", "goal": "patterns"}
```

```
print('{data[tool]} try to find {data[goal]} in the dataset'.format(data=my_methods))
```

```
Unsupervised algorithms try to find patterns in the dataset
```

Format specifier

- Specify data type to be used: `{index:specifier}`

```
print("Only {0:f}% of the {1} produced worldwide is {2}!".format(0.5155675, "data", "analyzed"))
```

```
Only 0.515568% of the data produced worldwide is analyzed!
```

```
print("Only {0:.2f}% of the {1} produced worldwide is {2}!".format(0.5155675, "data", "analyzed"))
```

```
Only 0.52% of the data produced worldwide is analyzed!
```

Formatting datetime

```
from datetime import datetime
print(datetime.now())
```

```
datetime.datetime(2019, 4, 11, 20, 19, 22, 58582)
```

```
print("Today's date is {:%Y-%m-%d %H:%M}".format(datetime.now()))
```

```
Today's date is 2019-04-11 20:20
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Formatted string literal

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

f-strings

- Minimal syntax
- Add prefix `f` to string

f"literal string {expression}"

```
way = "code"
method = "learning Python faster"
print(f"Practicing how to {way} is the best method for {method}")
```

```
Practicing how to code is the best method for learning Python faster
```

Type conversion

- Allowed conversions:
  - `!s` (string version)
  - `!r` (string containing a printable representation, i.e. with quotes)
  - `!a` (some as !r but escape the non-ASCII characters)

```
name = "Python"
print(f"Python is called {name!r} due to a comedy series")
```

```
Python is called 'Python' due to a comedy series
```

Format specifiers

- Standard format specifier:
  - `e` (scientific notation, e.g. 5 10^3)
  - `d` (digit, e.g. 4)
  - `f` (float, e.g. 4.5353)

```
number = 90.41890417471841
print(f"In the last 2 years, {number:.2f}% of the data was produced worldwide!")
```

```
In the last 2 years, 90.42% of the data was produced worldwide!
```

Format specifiers

- `datetime`

```
from datetime import datetime
my_today = datetime.now()
```

```
print(f"Today's date is {my_today:%B %d, %Y}")
```

```
Today's date is April 14, 2019
```

Index lookups

```
family = {"dad": "John", "siblings": "Peter"}
```

```
print("Is your dad called {family[dad]}?".format(family=family))
```

```
Is your dad called John?
```

- Use quotes for index lookups: `family["dad"]`

```
print(f"Is your dad called {family[dad]}?")
```

```
NameError: name 'dad' is not defined
```

Escape sequences

- Escape sequences: backslashes `\`

```
print("My dad is called "John"")
```

```
SyntaxError: invalid syntax
```

```
my_string = "My dad is called \"John\""
```

```
My dad is called "John"
```

Escape sequences

```
family = {"dad": "John", "siblings": "Peter"}
```

- Backslashes are not allowed in f-strings

```
print(f"Is your dad called {family[\"dad\"]}?")
```

```
SyntaxError: f-string expression part cannot include a backslash
```

```
print(f"Is your dad called {family['dad']}?")
```

```
Is your dad called John?
```

Inline operations

- Advantage: evaluate expressions and call functions inline

```
my_number = 4
my_multiplier = 7
```

```
print(f'{my_number} multiplied by {my_multiplier} is {my_number * my_multiplier}')
```

```
4 multiplied by 7 is 28
```

Calling functions

```
def my_function(a, b):
  return a + b
```

```
print(f"If you sum up 10 and 20 the result is {my_function(10, 20)}")
```

```
If you sum up 10 and 20 the result is 30
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Template method

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Template strings

- Simpler syntax
- Slower than f-strings
- Limited: don't allow format specifiers
- Good when working with externally formatted strings

Basic syntax

```
from string import Template
my_string = Template('Data science has been called $identifier')
my_string.substitute(identifier="sexiest job of the 21st century")
```

```
'Data science has been called sexiest job of the 21st century'
```

Substitution

- Use many `$identifier`
- Use variables

```
from string import Template
job = "Data science"
name = "sexiest job of the 21st century"
my_string = Template('$title has been called $description')
my_string.substitute(title=job, description=name)
```

```
'Data science has been called sexiest job of the 21st century'
```

Substitution

- Use `${identifier}` when valid characters follow identifier

```
my_string = Template('I find Python very ${noun}ing but my sister has lost $noun')
my_string.substitute(noun="interest")
```

```
'I find Python very interesting but my sister has lost interest'
```

Substitution

- Use `$$` to escape the dollar sign

```
my_string = Template('I paid for the Python course only $$ $price, amazing!')
my_string.substitute(price="12.50")
```

```
'I paid for the Python course only $ 12.50, amazing!'
```

Substitution

- Raise error when placeholder is missing

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
my_string.substitute(favorite)
```

```
Traceback (most recent call last):
KeyError: 'cake'
```

Substitution

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
```

```
try:
    my_string.substitute(favorite)
except KeyError:
      print("missing information")
```

```
missing information
```

Safe substitution

- Always tries to return a usable string
- Missing placeholders will appear in resulting string

```
favorite = dict(flavor="chocolate")
my_string = Template('I love $flavor $cake very much')
my_string.safe_substitute(favorite)
```

```
'I love chocolate $cake very much'
```

Which should I use?

- `str.format()` :
  - Good to start with. Concepts apply to f-strings.
  - Compatible with all versions of Python.
- **f-strings**:
  - Always advisable above all methods.
  - Not suitable if not working with modern versions of Python (3.6+).
- **Template strings**:
  - When working with external or user-provided strings

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Introduction to regular expressions

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

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}'
```

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*

- Normal characters match themselves (`st`)

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*

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

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*

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

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*

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

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*

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

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

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']
```

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', '']
```

The re module

```
import re
```

- Replace one or many matches with a string:

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

```
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']
```

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']
```

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'
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Repetitions

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Science

datacamp

Repeated characters

Validate the following string:

Repeated characters

Validate the following string:

password1234

Repeated characters

Validate the following string:

password1234

Repeated characters

Validate the following string:


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

```
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'>
```

Repeated characters

Validate the following string:


```
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)
```

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']
```

Quantifiers

- Zero times or more: `*`

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

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

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']
```

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)
```

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)
```

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)
```

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']
```

Quantifiers

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

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Regex metacharacters

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

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
```

Special characters

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


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

Special characters

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


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

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

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']
```

Special characters

- End of the string: `$`

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

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

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

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.']
```

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']
```

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']
```

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']
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Greedy vs. non-greedy matching

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

Greedy vs. non-greedy matching

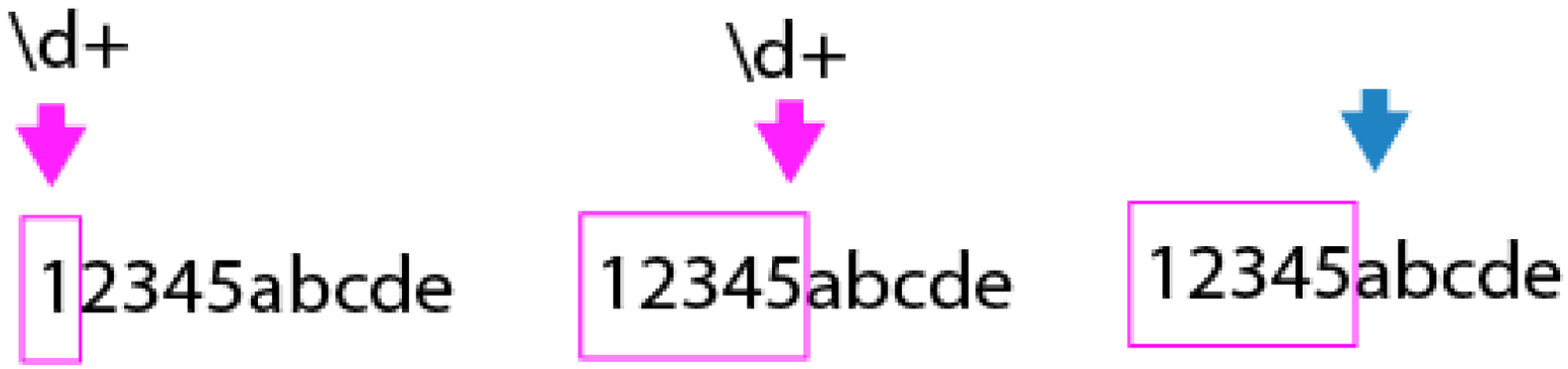
- Two types of matching methods:
  - Greedy
  - Non-greedy or lazy
- Standard quantifiers are greedy by default: `*`, `+`, `?`, `{num, num}`

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'>
```

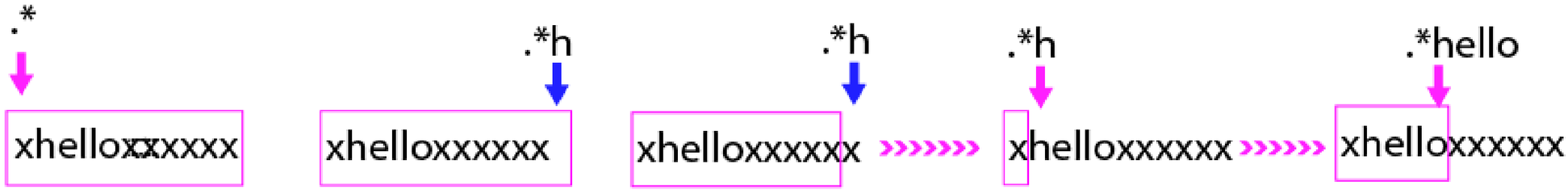


Greedy matching

- 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'>
```

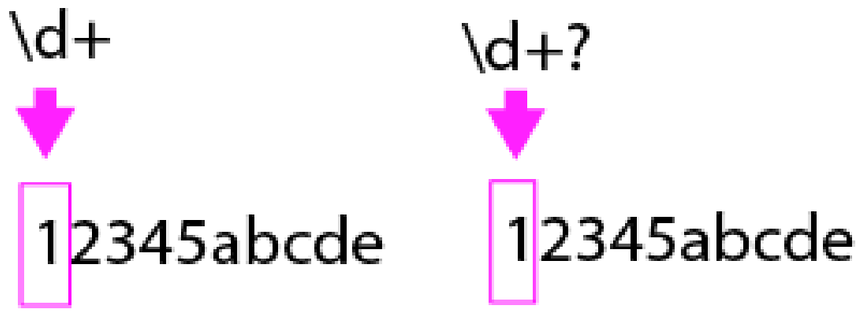


Non-greedy matching

- **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'>
```



Non-greedy matching

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

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

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

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Grouping and capturing

REGULAR EXPRESSIONS IN PYTHON

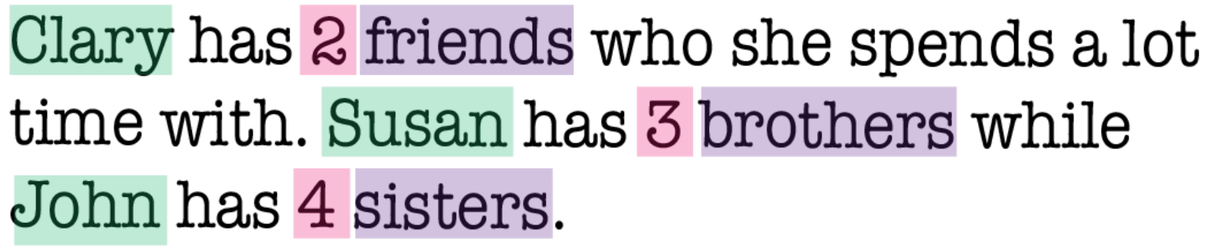
Maria Eugenia Inzaugarat
Data Scientist

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




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

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

Capturing groups

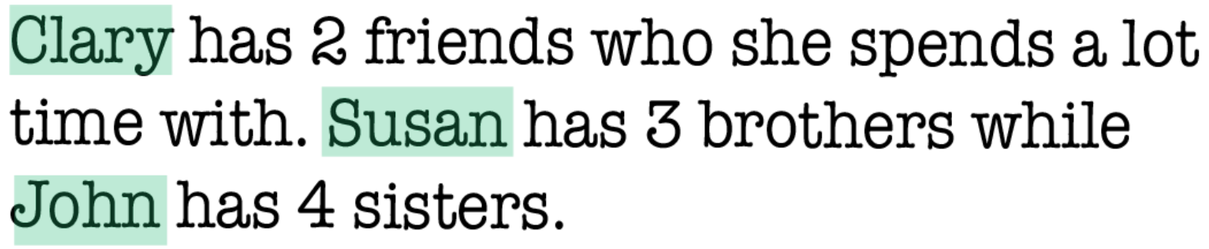
Clary has 2 friends who she spends a lot time with. Susan has 3 brothers while John has 4 sisters.

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

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

Group

Capturing groups



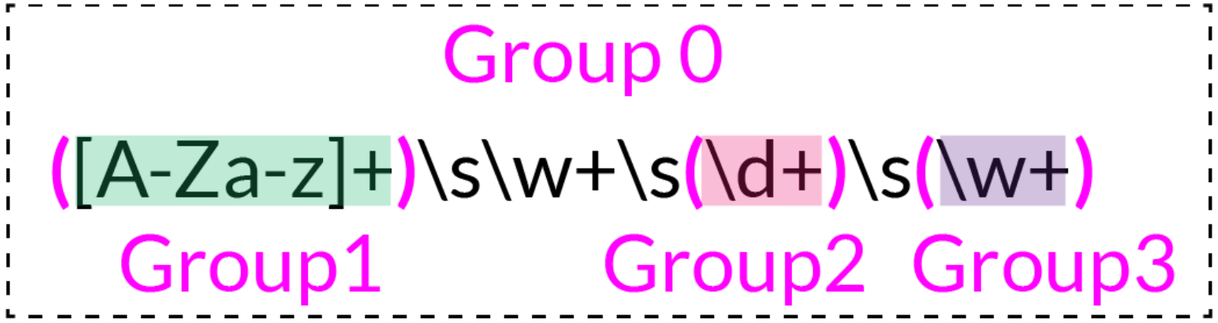
- Use parentheses to group and capture characters together

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

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

Capturing groups

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



Capturing groups

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

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

```
'Clary'
```

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'>
```

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']
```

```
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)
```

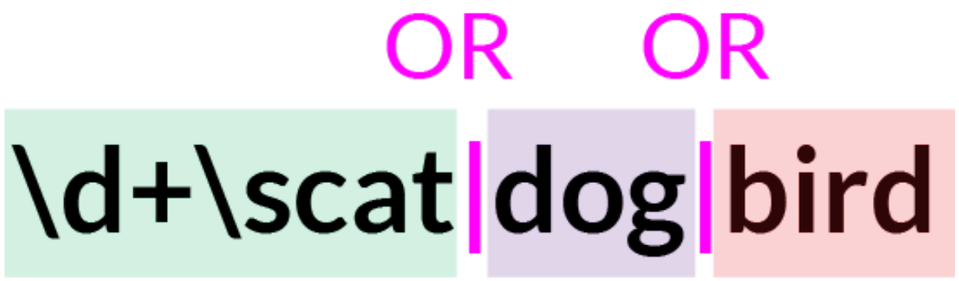
```
['cat', 'dog', 'bird']
```

Pipe

- Vertical bar or pipe: `|`

```
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']
```

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')]
```

Non-capturing groups

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

Non-capturing groups

- Match but **not capture** a group

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

Group1

```
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']
```

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']
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Backreferences

REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

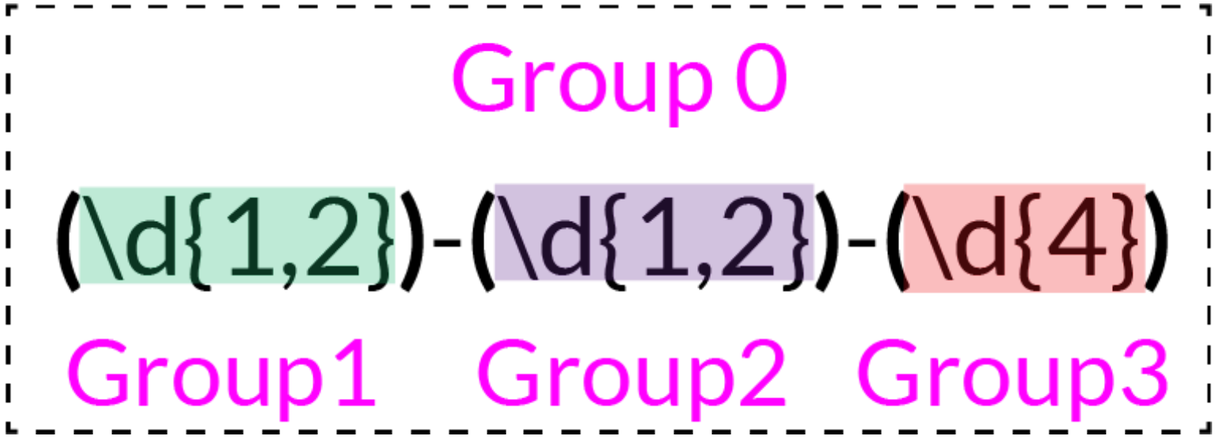
Numbered groups

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})
```

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.



Numbered groups

```
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

(?P<name>regex)

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")
```

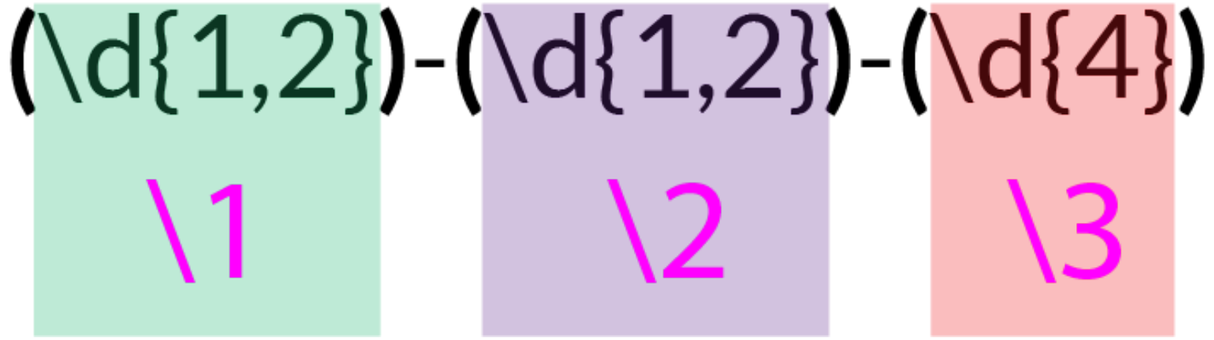
```
'Austin'
```

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

```
'78701'
```

Backreferences

- Using capturing groups to reference back to a group



Backreferences

- Using numbered capturing groups to reference back

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

Backreferences

- Using numbered capturing groups to reference back

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

```
['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']
```

Backreferences

- Using named capturing groups to reference back

(?P<name>regex)

\g<name>

```
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

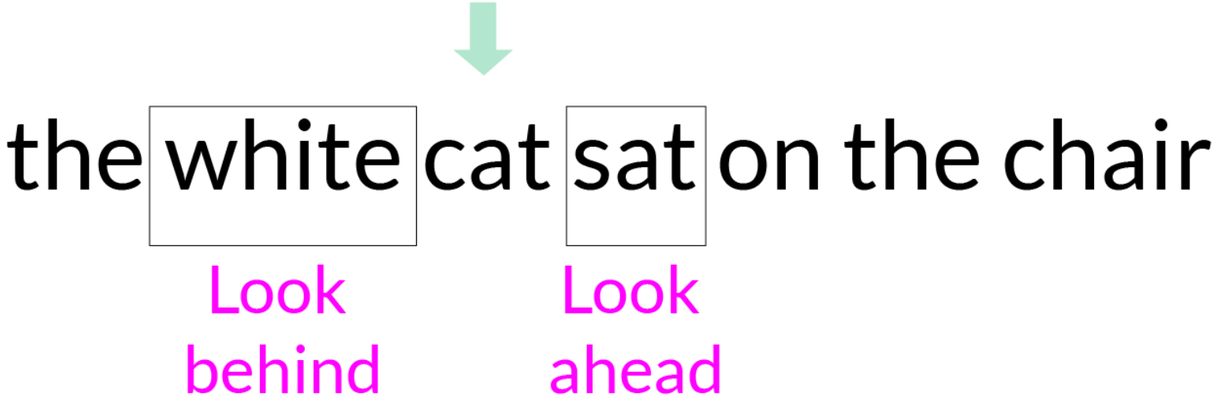
Looking around

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

the white cat sat on the chair

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.

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)

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)
```

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']
```

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)
```

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']
```

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)

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)
```

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']
```

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']
```

Let's practice!

REGULAR EXPRESSIONS IN PYTHON

Finishing line

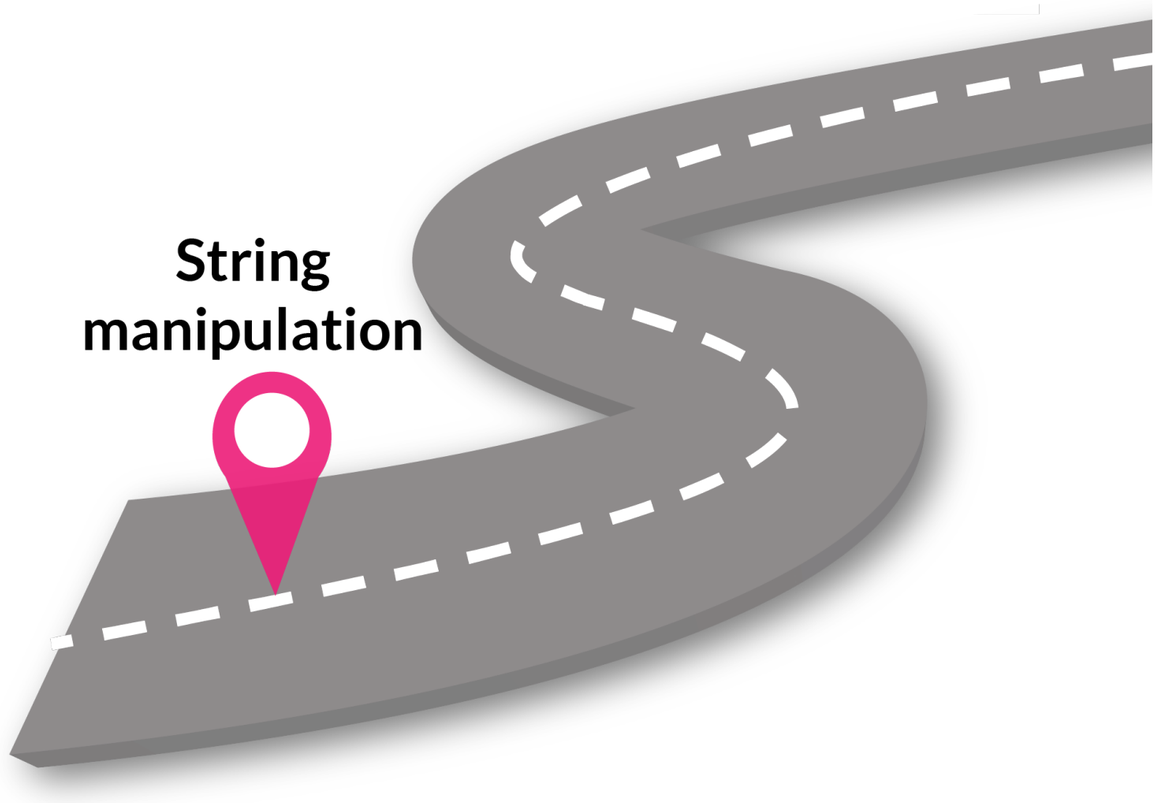
REGULAR EXPRESSIONS IN PYTHON

Maria Eugenia Inzaugarat
Data Scientist

datacamp

r”(Congratulations!)+”

Our journey



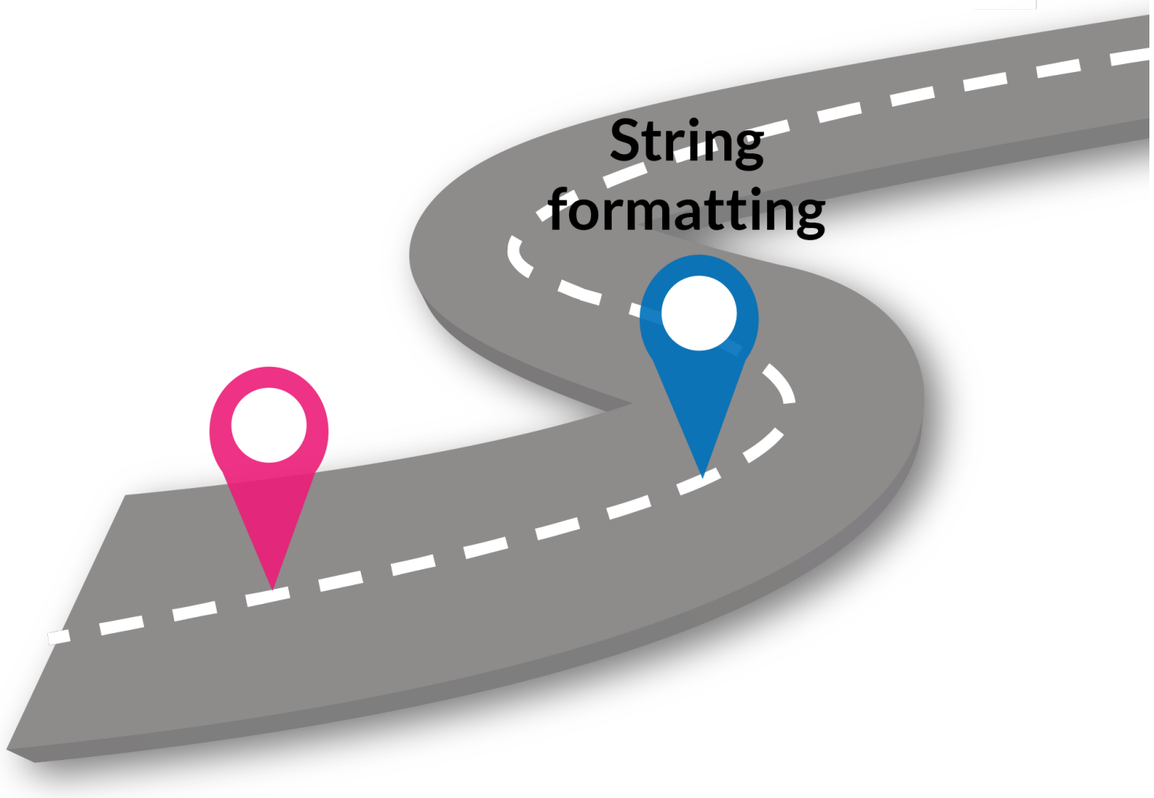
Key concepts

Concatenate and split

Index and slice strings

Replace and remove characters

Our journey

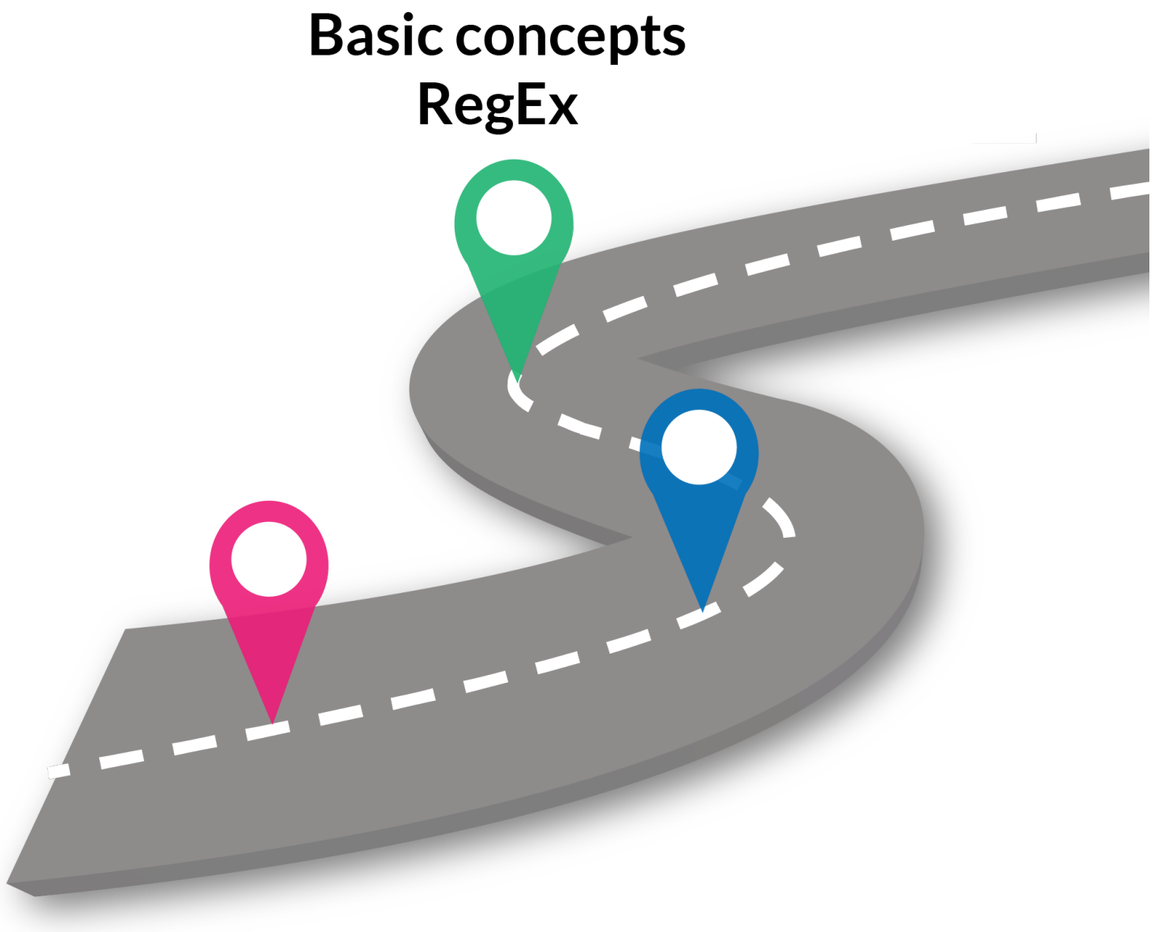


Insert custom strings into a predefined text

Three string formatting methods

Best approach according to situation

Our journey



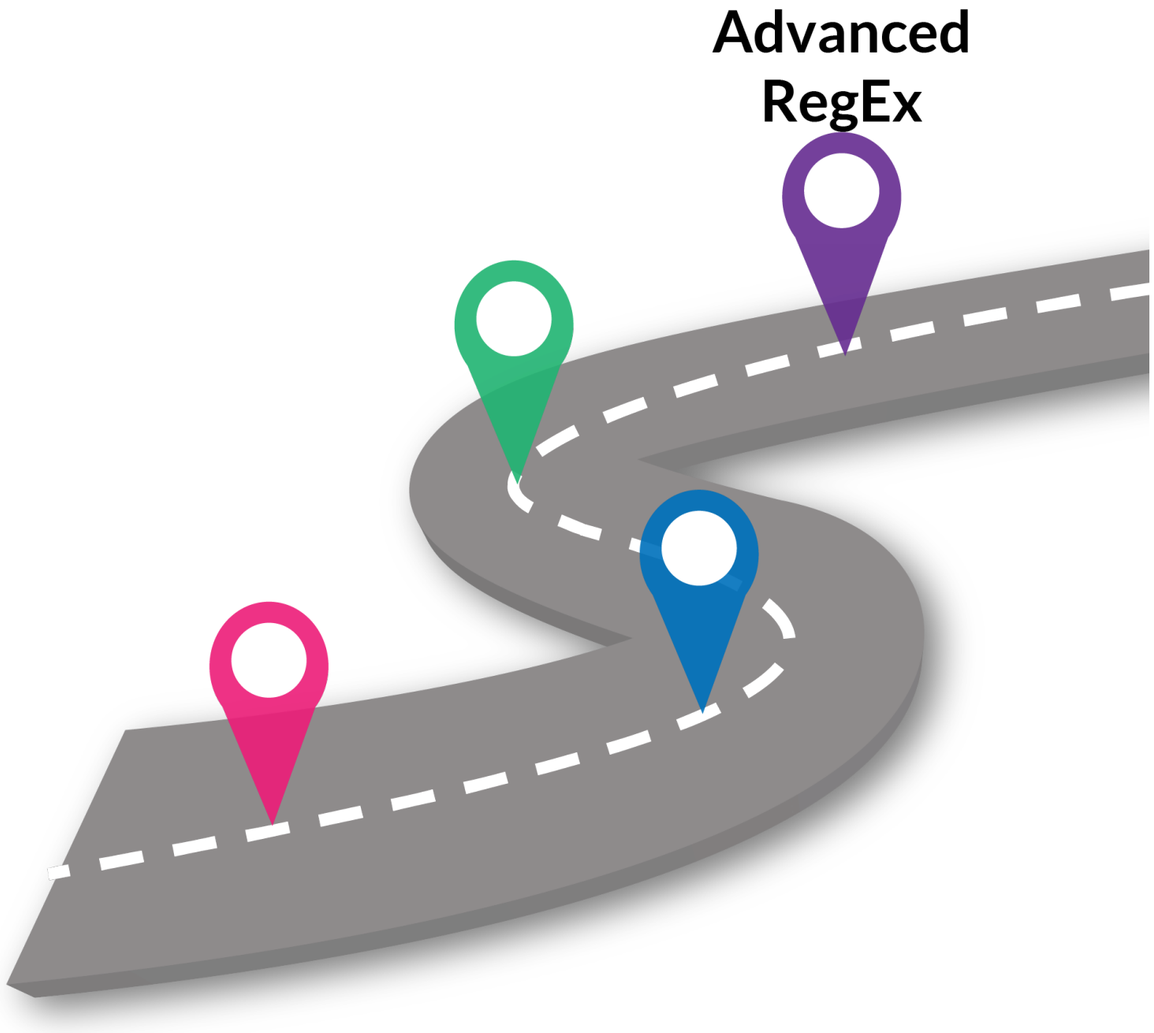
Basic syntax

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