https://campus.datacamp.com/courses/advanced-nlp-with-spacy/processing-pipelines?ex=1

03.01

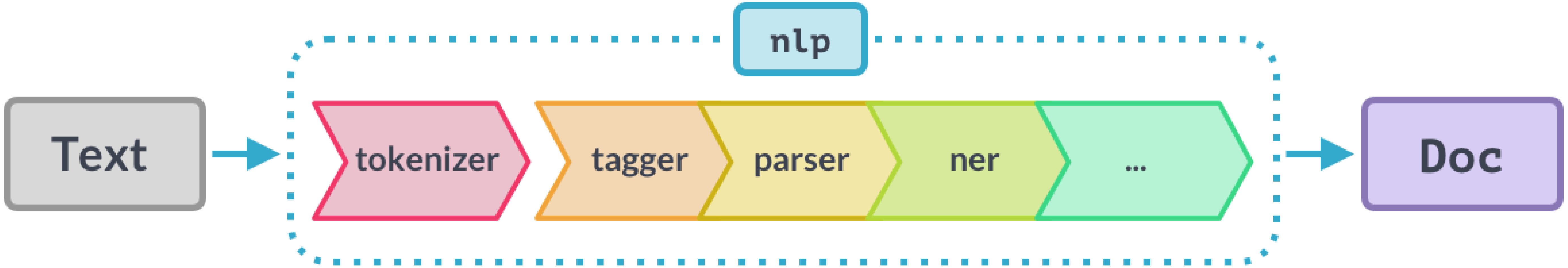
Processing pipelines

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

What happens when you call nlp?

Processing pipelines: series of functions applied to a Doc to add attributes like part-of-speech tags, dependency labels or named entities.




```
doc = nlp("This is a sentence.")
```

What does the nlp object actually do?
First, the tokenizer is applied to turn the string of text into a Doc object.
Next, a serie of pipeline components is applied to the Doc in order.
In this case, the tagger, then the parser, then the enitity recognizer.
Finally, the processed Doc is returned, so you can work with it.

Built-in pipeline components

spaCy ships with the following built-in pipeline components.

Base noun phrases, also known as noun chunks.

Name	Description	Creates
tagger	Part-of-speech tagger	`Token.tag`
parser	Dependency parser	`Token.dep` , `Token.head` , `Doc.sents` , `Doc.noun_chunks`
ner	Named entity recognizer	`Doc.ents` , `Token.ent_iob` , `Token.ent_type`
textcat	Text classifier	`Doc.cats`

Finally, the text classifier sets category labels that apply to the whole text, and adds them to the doc.cats property. Because text categories are always very specific, the text classifier is not included in any of the pre-trained models by default. But you can use it to train your own system.

Under the hood

All models you can load into spaCy include several files and a meta JSON.
The meta defines things like the language and pipeline.
This tells spaCy which componentes to instantiate.
The built-in components that make predictions also need binary data. The data is included in the model packaage and loaded into the component when you load the model.

meta.json

```
{
  "lang":"en",
  "name":"core_web_sm",
  "pipeline":["tagger", "parser", "ner"]
}
```

- Pipeline defined in model's `meta.json` in order
- Built-in components need binary data to make predictions

Pipeline attributes

- `nlp.pipe_names` : list of pipeline component names

```
print(nlp.pipe_names)
```

To see the names of the pipeline components present in the current nlp object, you can use the nlp.pipe names attribute.

```
['tagger', 'parser', 'ner']
```

- `nlp.pipeline` : list of `(name, component)` tuples

```
print(nlp.pipeline)
```

For a list of component name and component function, you can use the nlp.pipeline attirbute.
The component functions are functions applied to the Doc to process it and set attributes.

For example, part-of-speech tags or named entities.

```
[('tagger', <spacy.pipeline.Tagger>),
 ('parser', <spacy.pipeline.DependencyParser>),
 ('ner', <spacy.pipeline.EntityRecognizer>)]
```

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/processing-pipelines?ex=4

03.04

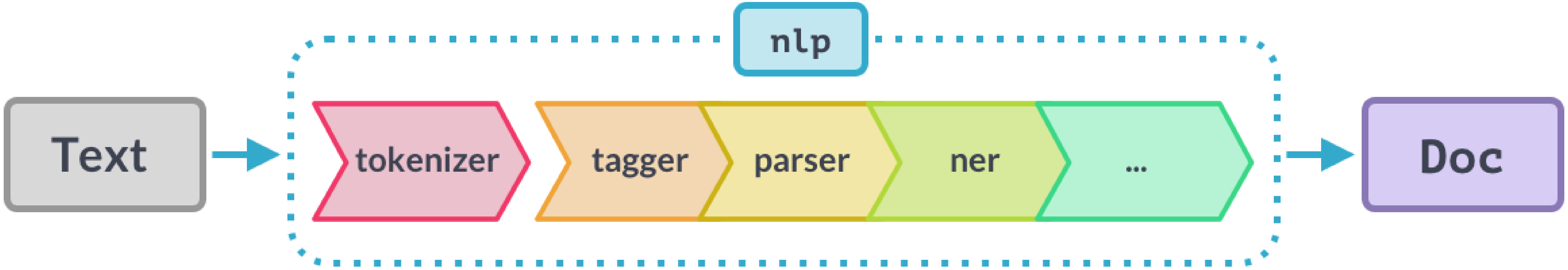
Custom pipeline components

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Why custom components?

Custom pipeline components let you add your own function to the spaCy pipeline that is executed when you call the nlp object on a text - for example, to modify the Doc and add more data to it.
After the text is tokenized and a Doc object has been created, pipeline components are applied in order.



spaCy supports a rage of built-in components, but also lets you define your own.

- Make a function execute automatically when you call `nlp`
- Add your own metadata to documents and tokens
- Updating built-in attributes like `doc.ents`

Anatomy of a component (1)

Fundamentally, a pipeline component is a function or callable that takes a doc, modifies it and returns it, so it can be processed by the next component in the pipeline.

Components can be added to the pipeline using the nlp.add_pipe method.

The method takes at least one argument: the comonent function.

- Function that takes a `doc`, modifies it and returns it
- Can be added using the `nlp.add_pipe` method

```
def custom_component(doc):
    # Do something to the doc here
    return doc

nlp.add_pipe(custom_component)
```

Anatomy of a component (2)

```
def custom_component(doc):
    # Do something to the doc here
    return doc

nlp.add_pipe(custom_component)
```

To specify "where" to add the component in the pipeline, you can use the following:

"last" is the default.

"first" will add the component first in the pipeline, right after the tokenizer.

Argument	Description	Example
`last`	If `True` , add last	`nlp.add_pipe(component, last=True)`
`first`	If `True` , add first	`nlp.add_pipe(component, first=True)`
`before`	Add before component	`nlp.add_pipe(component, before='ner')`
`after`	Add after component	`nlp.add_pipe(component, after='tagger')`

Example: a simple component (1)

We start iff with the small English model.

```
# Create the nlp object
nlp = spacy.load('en_core_web_sm')

# Define a custom component
def custom_component(doc):

    # Print the doc's length
    print('Doc length:' len(doc))

    # Return the doc object
    return doc

# Add the component first in the pipeline
nlp.add_pipe(custom_component, first=True)

# Print the pipeline component names
print('Pipeline:', nlp.pipe_names)
```

When we print the pipeline component names, the custom component now shows up at the start.
This means it will be applied first when we process a Doc.

```
Pipeline: ['custom_component', 'tagger', 'parser', 'ner']
```

Example: a simple component (2)

```
# Create the nlp object
nlp = spacy.load('en_core_web_sm')

# Define a custom component
def custom_component(doc):

    # Print the doc's length
    print('Doc length:' len(doc))

    # Return the doc object
    return doc

# Add the component first in the pipeline
nlp.add_pipe(custom_component, first=True)
# Process a text
doc = nlp("Hello world!")
```

Now when we process a text using the nlp object, the custom component will be applied to the Doc and the length of the document will be printed.

```
Doc length: 3
```

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/processing-pipelines?ex=8

03.08

Extension attributes

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Setting custom attributes

In this video, you'll learn how to add custom attributes to the Doc, Token and Span objects to store custom data.

Custom attributes let you add any meta data to Docs, Tokens and Spans.
The data can be added once, or it can be computed dynamically.
Custom attributes are available via ._. property.

This makes it clear that they were added by the user, and not built into spaCy, like token.text.

Attributes need to be registered on the global Doc, Token and Span classes you can import from spacy.tokens.

- Add custom metadata to documents, tokens and spans
- Accessible via the `._` property

```
doc._.title = 'My document'
token._.is_color = True
span._.has_color = False
```

- registered on the global `Doc`, `Token` or `Span` using the `set_extension` method

```
# Import global classes
from spacy.tokens import Doc, Token, Span
# Set extensions on the Doc, Token and Span
Doc.set_extension('title', default=None)
Token.set_extension('is_color', default=False)
Span.set_extension('has_color', default=False)
```

keyword argument let you define how the value should be computed. In this case, it has a default value and can be overwritten.

Extension attribute types

1. Attribute extensions
2. Property extensions
3. Method extensions

Attribute extensions

- Set a default value that can be overwritten

```
from spacy.tokens import Token

# Set extension on the Token with default value
Token.set_extension('is_color', default=False)
doc = nlp("The sky is blue.")

# Overwrite extension attribute value
doc[3]._.is_color = True
```

Property extensions (1)

- Define a getter and an optional setter function
- Getter only called when you *retrieve* the attribute value

```
from spacy.tokens import Token

# Define getter function
def get_is_color(token):
    colors = ['red', 'yellow', 'blue']
    return token.text in colors
# Set extension on the Token with getter
Token.set_extension('is_color', getter=get_is_color)
doc = nlp("The sky is blue.")
print(doc[3]._.is_color, '-', doc[3].text)
```

```
blue - True
```

Property extensions (2)

If you want to set extension attributes on a Span, you almost always want to use a property extension with a getter. Otherwise, you'd have to update "every possible span ever" by hand to set all the values.

- `Span` extensions should almost always use a getter

```
from spacy.tokens import Span

# Define getter function
def get_has_color(span):
    colors = ['red', 'yellow', 'blue']
    return any(token.text in colors for token in span)
# Set extension on the Span with getter
Span.set_extension('has_color', getter=get_has_color)
doc = nlp("The sky is blue.")
print(doc[1:4]._.has_color, '-', doc[1:4].text)
print(doc[0:2]._.has_color, '-', doc[0:2].text)
```

```
True - sky is blue
False - The sky
```

Method extensions

Method extensions make the extension attribute a callable method.
You can then pass one or more arguments to it, and compute attribute values dynamically, e. based on a certain argument or setting.
In this example, the method function checks whether the doc contains a token with a given text.

- Assign a **function** that becomes available as an object method
- Lets you pass **arguments** to the extension function

```python
from spacy.tokens import Doc

# Define method with arguments
def has_token(doc, token_text):
    in_doc = token_text in [token.text for token in doc]
# Set extension on the Doc with method
Doc.set_extension('has_token', method=has_token)
doc = nlp("The sky is blue.")
print(doc._.has_token('blue'), '- blue')
print(doc._.has_token('cloud'), '- cloud')
```

```
True - blue
False - cloud
```

Let's practice!

ADVANCED NLP WITH SPACY

https://campus.datacamp.com/courses/advanced-nlp-with-spacy/processing-pipelines?ex=13

03.13

Scaling and performance

ADVANCED NLP WITH SPACY

Ines Montani
spaCy core developer

Processing large volumes of text

- Use `nlp.pipe` method
- Processes texts as a stream, yields `Doc` objects
- Much faster than calling `nlp` on each text

BAD:

```
docs = [nlp(text) for text in LOTS_OF_TEXTS]
```

GOOD:

```
docs = list(nlp.pipe(LOTS_OF_TEXTS))
```

In this video, I'll show you a few tips and tricks to make your spaCy pipelines ru as fast as possible, and process large volumes of text efficiently.

If you need to process a lot of texs and create a lot of Doc objects in a row, the nlp.pipe method can speed this up significantly.

It processes the texts as a stream and yields Doc objects.
It is much faster than just calling nlp on each text, because it batches up the texts.
nlp.pipe is a generator that yields Doc objects, so in order to get a list of Docs, remember to call the list method around it.

Passing in context (1)

nlp.pipe also supports passing in tuples of text/context if you set "as tuples" to True.
The method will then yield doc/context tuples.

- Setting `as_tuples=True` on `nlp.pipe` lets you pass in `(text, context)` tuples
- Yields `(doc, context)` tuples
- Useful for associating metadata with the `doc`

This is useful for passing in additional metadata, like an ID associated with the text, or a page number.

```
data = [
    ('This is a text', {'id': 1, 'page_number': 15}),
    ('And another text', {'id': 2, 'page_number': 16}),
]

for doc, context in nlp.pipe(data, as_tuples=True):
    print(doc.text, context['page_number'])
```

```
This is a text 15
And another text 16
```

Passing in context (2)

You can even add the context meta data to custom attributes.

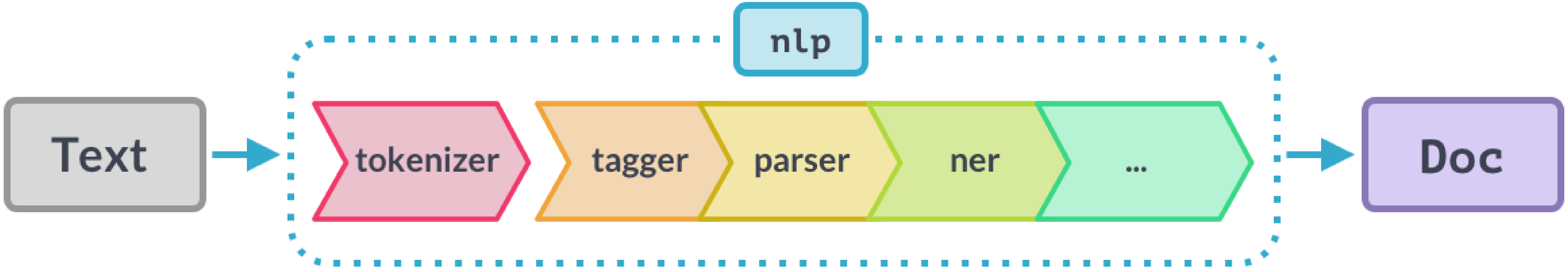
In this example, we are registering two extensions, "id" and "page number", which default to None.

After processing the text and passing through the context, we can overwrite the doc extensions with our context metadata.

```
from spacy.tokens import Doc

Doc.set_extension('id', default=None)
Doc.set_extension('page_number', default=None)

data = [
    ('This is a text', {'id': 1, 'page_number': 15}),
    ('And another text', {'id': 2, 'page_number': 16}),
]

for doc, context in nlp.pipe(data, as_tuples=True):
    doc._.id = context['id']
    doc._.page_number = context['page_number']
```

Using only the tokenizer

Another common scenario: Sometimes you already have a model loaded to do other processing, but you only need the tokenizer for one particular text.
Running the whole pipeline is unnecessarily slow, because you'll be getting a bunch of predictions from the model that you don't need.



- don't run the whole pipeline!

Using only the tokenizer (2)

If you only need a tokenized Doc object, you can use the nlp.make_doc method instead, which takes a text and returns a Doc.

This is also how spaCy does it behind the scenes: nlp.make_doc turns the text into a Doc before the pipeline components are called.

- Use `nlp.make_doc` to turn a text in to a `Doc` object

BAD:

```
doc = nlp("Hello world")
```

GOOD:

```
doc = nlp.make_doc("Hello world!")
```

Disabling pipeline components

- Use `nlp.disable_pipes` to temporarily disable one or more pipes

```
# Disable tagger and parser
with nlp.disable_pipes('tagger', 'parser'):
    # Process the text and print the entities
    doc = nlp(text)
    print(doc.ents)
```

  - restores them after the `with` block
  - only runs the remaining components

spaCy also allows you to temporarily disable pipeline components using the nlp.disable_pipes context manager.

It takes a variable number of arguments, the string names of the pipeline components to disable.

For example, if you only want to use the entity recognizer to process a document, you can temporarily disable the tagger and parser.

After the with block, the disabled pipeline components are automatically restored.

In the with block, spaCy will only run the remaining components.

Ok. It's your turn!
Let's try out the new methods and optimize some code to be faster and more efficient.

Let's practice!

ADVANCED NLP WITH SPACY