spaCy Cheat Sheet

> spaCy is a free, open-source library for advanced Natural Language processing (NLP) in Python. It’s designed specifically for production use and helps you build applications that process and understand large volumes of text. Documentation: spacy.io

>>> $ pip install spacy
>>> import spacy

> spaCy

> Statistical models

Download statistical models

Predict part-of-speech tags, dependency labels, named entities, and more. See here for available models: spacy.io/models

>>> $ python -m spacy download en_core_web_sm

Check that your installed models are up to date

>>> $ python -m spacy list

Loading statistical models

>>> import spacy
>>> nlp = spacy.load('en_core_web_sm')

> Documents and tokens

Processing text

Processing text with the nlp object returns a Doc object that holds all information about the tokens, their linguistic features and their relationships

>>> doc = nlp("This is a text")

Accessing token attributes

>>> doc[2].text
>>> doc[2].lemma
>>> doc[2].pos

> Label explanations

>>> space.english
>>> space.english

> Speech recognition

Accessing spans

Span indices are exclusive. So doc[2:4] is a span starting at token 2, up to — but not including — token 4.

>>> doc = nlp("This is a text")
>>> span = doc[2:4]
>>> span.text
"a text"

Creating a span manually

>>> from spacy.tokens import Span
>>> span = Span(doc, 2, 4, label="ENT")

Part-of-speech tags

Attributes return labels. For string labels, use the attributes with an underscore. For example, token._.pos

>>> doc = nlp("This is a text.")

Syntactic dependencies

Predicted by Statistical model

>>> doc = nlp("This is a text.")

Named entities

Predicted by Statistical model

>>> doc = nlp("This is a text.")

> Linguistic features

Pipeline components

Functions that take a Doc object, modify it and return it.

Text

Pipeline information

>>> nlp = spacy.load('en_core_web_sm')

>>> nlp.pipeline

["tagger", "parser", "ner"]

Custom components

def custom_component(doc): # Function that modifies the doc and returns it
print("Something to the doc!"")
return doc

nlp.add_pipe(custom_component, first=False) # Add the component first in the pipeline
Components can be added First, Last (inject), or before or after an existing component.

> Syntactic dependencies

Visualize dependencies

>>> doc = nlp("This is a sentence.")
>>> displacy.render(doc, style="dep")

Visualize named entities

>>> doc = nlp("Larry Page founded Google").ents

> Word vectors and similarity

To use word vectors, you need to load the larger models ending in "md" or "lg", for example en_core_web_md, en_core_web_lg...

Comparing similarity

>>> doc = nlp("Like cats")
>>> doc2 = nlp("Like dogs")

>>> doc.similarity(doc2)

Accessing word vectors

>>> doc = nlp("Like cats")
>>> doc.lemma_ # Lemmatize as a memory efficient way

>>> doc.vector

> Syntax iterators

Sentences

Usually needs the dependency parser

>>> doc = nlp("This is another one.")

Base noun phrases

Needs the tagger and parser

>>> doc = nlp("I have a red car")

After each chunk is a generator that yields spans

>>> doc.ents

[Chunk(text=Check in doc.ents, chunks) ("I", "a red car")]

Learn Data Skills Online at www.DataCamp.com

Learn spaCy online at www.DataCamp.com
# Attribute extensions

Attribute extensions are often used to enhance the functionality of classes. They allow for additional operations on objects. For example:

```python
# Example of using Attribute extensions

class Example:
    def __init__(self):
        self.color = 'red'

    def set_color(self, new_color):
        self.color = new_color

# Register a custom attribute on class
Example.register_attribute('custom', 'color', default='blue')

# Example usage
ex = Example()
ex.color = 'blue'
ex.set_color('green')
ex.custom
```

## Property extensions

Property extensions modify the behavior of properties. They can add or remove functionality from getting or setting properties.

```python
# Example of using Property extensions

class Example:
    def __init__(self):
        self.color = 'red'

    def set_color(self, new_color):
        self.color = new_color

# Register a custom property on class
Example.register_property('custom', 'color', default='blue')

# Example usage
ex = Example()
ex.color = 'blue'
ex.set_color('green')
ex.custom
```

## Method extensions

Method extensions can be used to add or modify methods of a class. They can add check functions, assertions, or other logic.

```python
# Example of using Method extensions

class Example:
    def __init__(self):
        self.color = 'red'

    def set_color(self, new_color):
        self.color = new_color

    def _check_color(self):
        assert self.color in ['red', 'green', 'blue'], 'Color must be red, green, or blue'

# Register a custom method on class
Example.register_method('check', '_check_color')

# Example usage
ex = Example()
ex.color = 'blue'
ex.set_color('green')
ex.check()
```

## Callable Method

Callable methods can be used to execute a method as a function. They can be used to call methods on objects.

```python
# Example of using Callable Method

class Example:
    def __init__(self):
        self.color = 'red'

    def set_color(self, new_color):
        self.color = new_color

# Register a custom Callable Method
Example.register_callable('set_color')

# Example usage
ex = Example()
ex.color = 'blue'
ex.set_color('green')
ex.set_color('blue')
```

---

## Rule-based matching

### Using the matcher

- **Matcher** is a class that supports adding or removing attributes.
- It can be used to add or remove attributes from a matcher.
- Matcher objects can be used to add or remove attributes from other objects.
- Matcher objects can be used to modify the behavior of other objects.

```python
# Example of using Matcher

class Matcher:
    def __init__(self):
        self.attributes = []

    def add_attribute(self, attribute):
        self.attributes.append(attribute)

    def remove_attribute(self, attribute):
        if attribute in self.attributes:
            self.attributes.remove(attribute)

    def modify_attribute(self, attribute, new_attribute):
        if attribute in self.attributes:
            self.attributes[self.attributes.index(attribute)] = new_attribute

# Example usage
matcher = Matcher()
matcher.add_attribute('color')
matcher.remove_attribute('color')
matcher.modify_attribute('color', 'green')
```

## Token patterns

Token patterns can be used to match specific tokens or patterns within text.

```python
# Example of using Token patterns

class TokenPattern:
    def __init__(self, pattern):
        self.pattern = pattern

    def match(self, text):
        return bool(re.search(self.pattern, text))

# Example usage
pattern = TokenPattern(r'\blove\b')
match = pattern.match('I love you')
```

## Operators and quantifiers

Operators and quantifiers can be used to modify the behavior of token patterns.

```python
# Example of using Operators and quantifiers

class TokenPattern:
    def __init__(self, pattern):
        self.pattern = pattern

    def match(self, text):
        return bool(re.search(self.pattern, text))

    def operator(self, operator):
        self.pattern = re.compile(operator + self.pattern)

# Example usage
pattern = TokenPattern(r'\blove\b')
match = pattern.match('I love you')
pattern.operator('+')
match = pattern.match('I love you')
```

## Glossary

- **Tokenizer**
- **Segmentation**
- **Lemmatization**
- **Sentence Boundary Detection**
- **POS Tagging**
- **Dependency Parsing**
- **Named Entity Recognition (NER)**
- **Text Classification**
- **Training**

---

**Learn Data Skills Online at**

[www.DataCamp.com](http://www.DataCamp.com)