Beyond Counting Words: Working with Word Embeddings

Damian Trilling

d.c.trilling@uva.nl
@damian0604
www.damiantrilling.net

12–13 April 2021

Afdeling Communicatiewetenschap
Universiteit van Amsterdam

This part: Recap "Working with textual data in Python"

Getting to know each other

Ways of working with data in Python

Functions and methods

Modifying lists and dictionaries

for, if/elif/else, try/except

Some examples of working with texts

Takeaways

Getting to know each other

Damian

dr. Damian Trilling

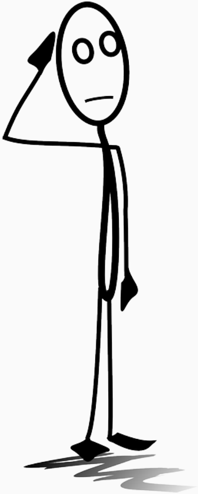
Universitair Hoofddocent (Associate Professor)

Communication in the Digital Society

- studied Communication Science in Münster and at the VU 2003–2009
- PhD candidate @ ASCoR 2009–2012
- political communication and journalism in a changing media environment
- computational research methods

@damian0604 d.c.trilling@uva.nl

REC-C 8th floor www.damiantrilling.net



Who are you, and what do you hope to get out of this workshop?

Pandas vs native

Pandas

Main data structure: the “dataframe”

- familiar; similar to R/SPSS/Stata
- great built-in methods for data wrangling
- we can easily apply operations to whole columns

[show in Notebook]

Pandas

However,

- your dataset may be too large to keep in memory;
- it may not make sense to think of your data as a table;
- you may not be interested in statistitical calculations *within* your dataframe (e.g., regress some column on some others).

⇒ Collections of texts that we want to use for Machine Learning are not necessarily something we want to keep in a table

Pandas

However,

- your dataset may be too large to keep in memory;
- it may not make sense to think of your data as a table;
- you may not be interested in statistitical calculations *within* your dataframe (e.g., regress some column on some others).

⇒ Collections of texts that we want to use for Machine Learning are not necessarily something we want to keep in a table

Pandas

However,

- your dataset may be too large to keep in memory;
- it may not make sense to think of your data as a table;
- you may not be interested in statistitical calculations *within* your dataframe (e.g., regress some column on some others).

⇒ Collections of texts that we want to use for Machine Learning are not necessarily something we want to keep in a table

Pandas

However,

- your dataset may be too large to keep in memory;
- it may not make sense to think of your data as a table;
- you may not be interested in statistitical calculations *within* your dataframe (e.g., regress some column on some others).

$\Rightarrow$ Collections of texts that we want to use for Machine Learning are not necessarily something we want to keep in a table

Pandas

However,

- your dataset may be too large to keep in memory;
- it may not make sense to think of your data as a table;
- you may not be interested in statistitical calculations *within* your dataframe (e.g., regress some column on some others).

$\Rightarrow$ Collections of texts that we want to use for Machine Learning are not necessarily something we want to keep in a table

Native data types

Lists and dictionaries

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Native data types

Lists and dictionaries

list `firstnames = ['Alice','Bob','Cecile']`
`lastnames = ['Garcia','Lee','Miller']`

list `ages = [18,22,45]`

dict `agedict = {'Alice': 18, 'Bob': 22, 'Cecile': 45}`

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Native data types

Lists and dictionaries

list `firstnames = ['Alice','Bob','Cecile']`
`lastnames = ['Garcia','Lee','Miller']`

list `ages = [18,22,45]`

dict `agedict = {'Alice': 18, 'Bob': 22, 'Cecile': 45}`

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Native data types

Lists and dictionaries

list
```
firstnames = ['Alice','Bob','Cecile']
lastnames = ['Garcia','Lee','Miller']
```
list
```
ages = [18,22,45]
```
dict
```
agedict = {'Alice': 18, 'Bob': 22,
'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Native data types

Lists and dictionaries

list `firstnames = ['Alice','Bob','Cecile']`
`lastnames = ['Garcia','Lee','Miller']`

list `ages = [18,22,45]`

dict `agedict = {'Alice': 18, 'Bob': 22, 'Cecile': 45}`

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Native datatypes

Retrieving specific items

list `firstnames[0]` gives you the first entry

`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18

Native datatypes

Retrieving specific items

list `firstnames[0]` gives you the first entry

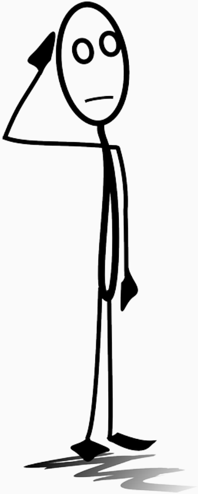
`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18



Think of at least two different ways of storing data about some fictious persons (first name, last name, age, phone number, . . .) using lists and/or dictionaries. What are the pros and cons?

Functions and methods

Python lingo

Functions

functions Take an input and return something else
`int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else
`int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with (). Between the (), *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else
`int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with (). Between the (), *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

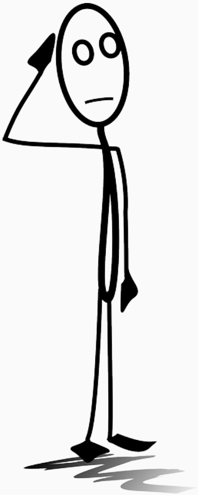
functions Take an input and return something else
`int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Some functions

```
len(x)        # returns the length of x
y = len(x)    # assign the value returned by len(x) to y
print(len(x)) # print the value returned by len(x)
print(y)      # print y
int(x)        # convert x to an integer
str(x)        # convert x to a string
sum(x)        # get the sum of x
```



How could you print the mean (average) of a list of integers using the functions on the previous slide?

Some methods

Some string methods

```
mystring = "Hi! How are you?"
mystring.lower() # return lowercased string (doesn’t change original!)
mylowercasedstring = mystring.lower() # save to a new variable
mystring = mystring.lower() # or override the old one
mystring.upper() # uppercase
mystring.split() # Splits on spaces and returns a list [’Hi!’, ’How’, ’are’, ’you?’]
```

We’ll look into some list methods later.

⇒ **You can use TAB-completion in Jupyter to see all methods (and properties) of an object!**

Writing own functions

You can write an own function:

```
def addone(x):
    y = x + 1
    return y
```

Functions take some input (“argument”) (in this example, we called it `x`) and *return* some result.

Thus, running

```
addone(5)
```

returns `6`.

Writing own functions

Attention, R users! (maybe obvious for others?)

You *cannot** apply the function that we just created on a whole list – after all, it takes an int, not a list as input.

(wait a sec foruntil we cover for loops later today, but this is how you'd do it (by calling the function for each element in the list separately):):

```
mynumbers = [5, 3, 2, 4]
results = [addone(e) for e in mynumbers]
```

* Technically speaking, you could do this by wrapping the `map` function around your own function, but that's not considered "pythonic". Don't do it ;-)

Modifying lists & dicts

Modifying lists

Let's use one of our first methods! Each *list* has a method `.append()`:

Appending to a list

```
mijnlijst = ["element 1", "element 2"]
anotherone = "element 3" # note that this is a string, not a list!
mijnlijst.append(anotherone)
print(mijnlijst)
```

gives you:

```
["element 1", "element 2", "element 3"]
```

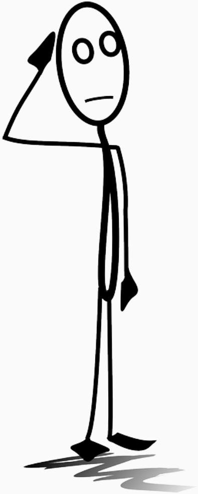
Modifying lists

Merging two lists (= extending)

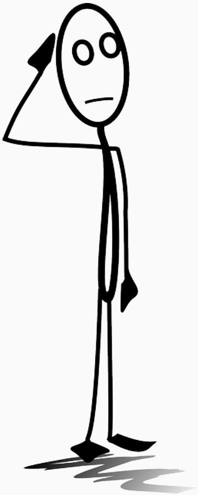
```
mijnlijst = ["element 1", "element 2"]
anotherone = ["element 3", "element 4"]
mijnlist.extend(anotherone)
print(mijnlijst)
```

gives you:

```
["element 1", "element 2", "element 3", "element 4]
```



What would have happened if we had used `.append()` instead of `.extend()`?



Why do you think that the Python developers implemented `.append()` and `.extend()` as methods of a list and not as functions?

Modifying dicts

Adding a key to a dict (or changing the value of an existing key)

```
mydict = {"whatever": 42, "something": 11}
mydict["somethingelse"] = 76
print(mydict)
```

gives you:

```
{'whatever': 42, 'somethingelse': 76, 'something': 11}
```

If a key already exists, its value is simply replaced.

for, if/elif/else, try/except

How can we structure our program?

If we want to *repeat* a block of code, exectute a block of code only *under specific conditions*, or more generally want to structure our code, we use *indention*.

Indention: The Python way of structuring your program

- Your program is structured by TABs or SPACEs.
- Jupyter (or your IDE) handles (guesses) this for you, but make sure to not interfere and not to mix TABs or SPACEs!
- Default: four spaces per level of indention.

Indention

Structure

A first example of an indented block – in this case, we want to *repeat* this block:

```
agedict = {'Zeus': None, 'Denis': 96, 'Alice': 18, 'Rebecca': 20 , 'Bob': 22, 'Cecile': 45}

myfriends = ['Alice','Bob','Cecile']

print ("The names and ages of my friends:")
for buddy in myfriends:
       print (f"My friend {buddy} is {agedict[buddy]} years old")
```

Output:

```
My friend Alice is 18 years old
My friend Bob is 22 years old
My friend Cecile is 45 years old
```

What happened here?

```
1 for buddy in myfriends:
2     print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The for loop

1. Take the first element from `myfriends` and call it `buddy` (like `buddy = myfriends[0]`) (line 1)
2. Execute the indented block (line 2, but could be more lines)
3. Go back to line 1, take next element (like `buddy = myfriends[1]`)
4. Execture the indented block . . .
5. . . . repeat until no elements are left . . .

The f-string (*formatted* string)

If you prepend a string with an `f`, you can use curly brackets texttt{} to insert the value of a variable

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` with indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

Can we also loop over dicts?

Sure! But we need to indicate how exactly:

```
mydict = {"A":100, "B": 60, "C": 30}

for k in mydict: # or mydict.keys()
    print(k)

for v in mydict.values():
    print(v)

for k,v in mydict.items():
    print(f"{k} has the value {v}")
```

Can we also loop over dicts?

The result:

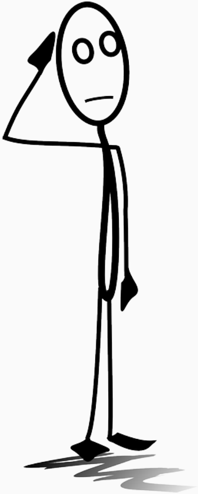
```
A
B
C

100
60
30

A has the value 100
B has the value 60
C has the value 30
```

if statements

Structure

Only execute block if condition is met

```
x = 5
if x <10:
  print(f"{x} is smaller than 10")
elif x > 20:
  print(f"{x} is greater than 20")
else:
  print("No previous condition is met, therefore 10<={x}<=20")
```



Can you see how such an if statement could be particularly useful when nested in a for loop?

try/except

Structure

If executed block fails, run another block instead

```
x = "5"
try:
  myint = int(x)
except:
  myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
mylist = ["5", 3, "whatever", 2.2]
myresults = []
for x in mylist:
    try:
        myresults.append(int(x))
    except:
        myresults.append(None)
print(myresults)
```

try/except

Structure

If executed block fails, run another block instead

```
x = "5"
try:
  myint = int(x)
except:
  myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
mylist = ["5", 3, "whatever", 2.2]
myresults = []
for x in mylist:
    try:
        myresults.append(int(x))
    except:
        myresults.append(None)
print(myresults)
```

List comprehensions

Structure

A for loop that `.append()`s to an empty list can be replaced by a one-liner:

```
mynumbers = [2,1,6,5]
mysquarednumbers = []
for x in mynumbers:
    mysquarednumbers.append(x**2))
```

is equivalent to:

```
mynumbers = [2,1,6,5]
mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
mynumbers = [2,1,6,5]
mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

Structure

A for loop that `.append()`s to an empty list can be replaced by a one-liner:

```
mynumbers = [2,1,6,5]
mysquarednumbers = []
for x in mynumbers:
    mysquarednumbers.append(x**2))
```

is equivalent to:

```
mynumbers = [2,1,6,5]
mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
mynumbers = [2,1,6,5]
mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

A very pythonic construct

- Every for loop can also be written as a for loop that appends to a new list to collect the results.
- For very complex operations (e.g., nested for loops), it can be easier to write out the full loops.
- But mostly, list comprehensions are really great! (and much more concise!)

$\Rightarrow$ **You really should learn this!**

Generators

Structure

A lazy for loop (or function) that only generates its next element when it is needed:

You can create a generator just like a list comprehension (but with `()` instead of `[]`):

```
mynumbers = [2,1,6,5]
squaregen = (x**2 for x in mynumbers) # these are NOT calculated yet
for e in squaregen:
    print(e)          # only here, we are calculating the NEXT item
```

Or like a function (but with `yield` instead of `return`):

```
def squaregen(listofnumbers):
    for x in listofnumbers:
        yield(x**2)
mygen = squaregen(mynumbers)
for e in mygen:
    print(e)
```

Generators

Structure

A lazy for loop (or function) that only generates its next element when it is needed:

You can create a generator just like a list comprehension (but with `()` instead of `[]`):

```
mynumbers = [2,1,6,5]
squaregen = (x**2 for x in mynumbers) # these are NOT calculated yet
for e in squaregen:
    print(e)           # only here, we are calculating the NEXT item
```

Or like a function (but with `yield` instead of `return`):

```
def squaregen(listofnumbers):
    for x in listofnumbers:
        yield(x**2)
mygen = squaregen(mynumbers)
for e in mygen:
    print(e)
```

Generators

A very memory and time efficient construct

- Every function that *returns* a list can also be written as a generator that *yields* the elements of the list
- Especially useful if
  - it takes a long time to calculate the list
  - the list is very large and uses a lot of memory (hi big data!)
  - the elements in the list are fetched from a slow source (a file, a network connection)
  - you don't know whether you actually will need all elements

Some examples of working with texts

Counting words

1. Split text into words ("tokenization")
2. Count words

Counting words

```
from collections import Counter
import re  # for alternative tokenization only

texts = [’This is the first text text text first’, ’And another text
     yeah yeah’]

# split on spaces
tokenized_texts = [t.split() for t in texts]
# alternative that splits on all "non-word" characters:
# tokenized_texts = [re.split(r"\W",t) for t in texts]

c = Counter(tokenized_texts[0])
print(c.most_common(3)

c2 = Counter(tokenized_texts[1])
print(c2.most_common(3))
```

```
(’text’, 3), (’first’, 2), (’This’, 1)]
[(’yeah’, 2), (’And’, 1), (’another’, 1)]
```

Some preprocessing

What do we have to improve?

lowercasing

```
texts2 = [t.lower() for t in texts]
```

removing punctuation (method 1)

```
texts3 = [t.replace('.','').replace(',','').replace('!','') for t in
    texts]
```

removing punctuation (method 2)

```
import string
trans = str.maketrans('', '', string.punctuation)
texts4 = [t.translate(trans) for t in texts]
```

Some preprocessing

What do we have to improve?

lowercasing

```
texts2 = [t.lower() for t in texts]
```

removing punctuation (method 1)

```
texts3 = [t.replace('.','').replace(',','').replace('!','') for t in
    texts]
```

removing punctuation (method 2)

```
import string
trans = str.maketrans('', '', string.punctuation)
texts4 = [t.translate(trans) for t in texts]
```

Stopword removal: What and why?

Why remove stopwords?

- If we want to identify key terms (e.g., by means of a word count), we are not interested in them
- In many analyses, irrelevant information will dominate the picture
- By removing them, we make our data and our models simpler and smaller

Stopword removal

e.g., with *list comprehension* and the `.join()`

```
mystopwords = ['he', 'her', 'a', 'one', 'the]
t ='He gives her a beer and a cigarette.'
t2 = " ".join([w for w in t.split() if w.lower() not in mystopwords])
```

t2 now is “gives beer cigarette”

ngrams

Instead of just looking at single words (unigrams), we can also use adjacent words (bigrams).

```
import nltk
texts = ['This is the first text text text first', 'And another text yeah yeah']
texts_bigrams = [["_".join(tup) for tup in nltk.ngrams(t.split(),2)] for t in texts]
print(texts_bigrams)
```

```
[['This_is', 'is_the', 'the_first', 'first_text',
'text_text', 'text_text', 'text_first'],
['And_another', 'another_text', 'text_yeah',
'yeah_yeah']]
```

Typically, we would combine both. What do you think? Why is this useful? (and what may be drawbacks?)

ngrams

Instead of just looking at single words (unigrams), we can also use adjacent words (bigrams).

```
import nltk
texts = ['This is the first text text text first', 'And another text
    yeah yeah']
texts_bigrams = [["_".join(tup) for tup in nltk.ngrams(t.split(),2)] for
     t in texts]
print(texts_bigrams)
```

```
[['This_is', 'is_the', 'the_first', 'first_text',
'text_text', 'text_text', 'text_first'],
['And_another', 'another_text', 'text_yeah',
'yeah_yeah']]
```

Typically, we would combine both. **What do you think? Why is this useful? (and what may be drawbacks?)**

NLP: What and why?

Why parse sentences?

- To find out what grammatical function words have
- and to get closer to the meaning.

Parsing a sentence

```
import nltk
sentence = "At eight o'clock on Thursday morning, Arthur didn't feel
    very good."
tokens = nltk.word_tokenize(sentence)
print (tokens)
```

`nltk.word_tokenize(sentence)` is similar to sentence.split(), but compare handling of punctuation and the `didn't` in the output:

```
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning','Arthur', 'did',
    "n't", 'feel', 'very', 'good', '.']
```

Parsing a sentence

Now, as the next step, you can "tag" the tokenized sentence:

```
tagged = nltk.pos_tag(tokens)
print (tagged[0:6])
```

gives you the following:

```
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

And you could get the word type of "morning" with `tagged[5][1]`!

Parsing a sentence

Now, as the next step, you can "tag" the tokenized sentence:

```
tagged = nltk.pos_tag(tokens)
print (tagged[0:6])
```

gives you the following:

```
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

And you could get the word type of "morning" with `tagged[5][1]`!

More NLP

Look at http://nltk.org

Look at http://spacy.io

Example: Named Entity Recognition with spacy

Terminal:

```
sudo pip3 install spacy
sudo python3 -m spacy download nl # or en, de, fr ....
```

Python:

```
import spacy
nlp = spacy.load('nl')
doc = nlp('De docent heet Damian, en hij geeft vandaag les. Daarnaast is
      hij een onderzoeker, net zoals Anne. Ze werken allebei op de UvA')
for ent in doc.ents:
    print(ent.text,ent.label_)
```

returns:

```
Damian MISC
Anne PER
UvA LOC
```

Example: Lemmatization

Lemmatization gives you the words in the form in which you would look them up in a good old dictionary.

```
import spacy
nlp = spacy.load('en')
doc = nlp("I am running while generously greeting my neighbors")
lemmatized = " ".join([word.lemma_ for word in doc])
print(lemmatized)
```

returns:

```
-PRON- be run while generously greet -PRON- neighbor
```

The last example, `spacy`, in fact uses models that are trained very much like the techniques we will discuss in this course.

Takeaways

Takeaways

- We can organize data either in dataframes (pandas) or in lists, dictionaries, or simular (native Python)
- There are methods (which are associated with an object) and functions (which are not). Methods are cool because we can discover them with tab completion
- If a function takes, say, a string as input, we cannot apply them to a list. But if we have a list of strings, we can use a list comprehension.
- We can structure our code with if/elif/else conditions, for loops, and try/except control structures

Takeaways

- We can organize data either in dataframes (pandas) or in lists, dictionaries, or simular (native Python)
- There are methods (which are associated with an object) and functions (which are not). Methods are cool because we can discover them with tab completion
- If a function takes, say, a string as input, we cannot apply them to a list. But if we have a list of strings, we can use a list comprehension.
- We can structure our code with if/elif/else conditions, for loops, and try/except control structures

Takeaways

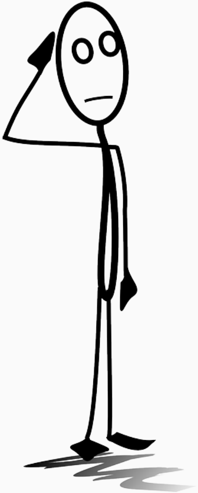
- We can organize data either in dataframes (pandas) or in lists, dictionaries, or simular (native Python)
- There are methods (which are associated with an object) and functions (which are not). Methods are cool because we can discover them with tab completion
- If a function takes, say, a string as input, we cannot apply them to a list. But if we have a list of strings, we can use a list comprehension.
- We can structure our code with if/elif/else conditions, for loops, and try/except control structures

Takeaways

- We can organize data either in dataframes (pandas) or in lists, dictionaries, or simular (native Python)
- There are methods (which are associated with an object) and functions (which are not). Methods are cool because we can discover them with tab completion
- If a function takes, say, a string as input, we cannot apply them to a list. But if we have a list of strings, we can use a list comprehension.
- We can structure our code with if/elif/else conditions, for loops, and try/except control structures

Takeaways

- There is something called "generator" that you can loop over like a list, but the next item is only generated once it is used. This way, you can process data that are larger than your memory, and you can start processing before all data are in memory.



Everybody up to speed? Ready to get started?