A short summary of todays session about iterators

Assume we have a class `Sentence` which will consist of words. If we write the class like this it will not be iterable like e.g. a `list`.

```
class Sentence:
    def __init__(self, text):
        self.words = text.split()

s = Sentence("the quick brown fox")
for word in s:
    print(word)
```

If we execute the code above we will get a `TypeError`. Our object is not `iterable`.

So, how can we make a class iterable?

To make a class `iterable` it must implement the `dunder method` `__iter__()` which must return an iterator.

What is an iterator, you may ask? An iterator is an object through which we can access elements of another object. In our sentence example the iterator is an object, which will deliver us words from our sentence object.

```
class Sentence:
    def __init__(self, text):
        self.words = text.split()

    def __iter__(self):
        """ Lets implement the iter method to make our sentence iterable
"""
        return iter(self.words)

s = Sentence("the quick brown fox")
for word in s:
    print(word)
```

The easiest way to do this is to use an iterator from a existing structure from which we know that we can iterate over. In our example we have the attribute `words` which is a list of words from our sentence. We can get an iterator from the list by calling the builtin function `iter()` on our `self.words` list.

With this simple implementation we can make our `Sentence` `iterable` and iterate over it.

What exactly is an iterator?

An iterator is an object fullfilling the `iterator protocol` as described here. The iterator protocols requires us to implement two methods: `__next__` and `__iter__`.

If we would create an iterator from scratch for our `Sentence` type it would propably look like this:

```
import re

class SentenceIterator:
    """ Didactic example: Implementation of the iterator protocol for a
sentence type as described in the GoF book.
        You would propably never do this in production code.
    """
    def __init__(self, sentence):
        self.words = sentence.words
        self.pos = 0

    def __next__(self):
        try:
            word = self.words[self.pos]
            self.pos += 1
            return word
        except IndexError:
            raise StopIteration()

    def __iter__(self):
        return self


class Sentence:
    RE_WORD = re.compile("\w+")

    def __init__(self, sentence):
        self.words = Sentence.RE_WORD.findall(sentence)

    def __iter__(self):
        return SentenceIterator(self)

s = Sentence("the quick brown fox")
for word in s:
    print(word)
```

Using generators as iterators

Generators behave like iterators so we can use them everywhere we need an iterator.

We can create generators using `generator function` or `generator expressions`. In this example we will just focus on generator functions.

Generator functions

- the presence of yield makes a function a generator function
- the function will not yield the return value directly, instead creates a generator object
- a generator object behaves like an iterator

- a generator object is exausted after usage

Using a generator function as iterator would look like this:

```
import re
RE_WORD = re.compile("\w+")

class Sentence:
    def __init__(self, sentence):
    self.words = RE_WORD.findall(sentence)

    def __iter__(self):
    # every function that contains a yield is automatically a generator
function
        for word in self.words:
            yield word


s = Sentence("the quick brown fox")
    for word in s:
    print(word)
```