Representing Documents; Unit Testing II

Benjamin Roth

CIS LMU

Documents and Word Statistics

- Often, documents are the units a natural language processing system starts with.
- Document: the basic organizational unit that is read in before further processing.
- *"Documents"* can be
  - Tweets
  - Wikipedia articles
  - Product reviews
  - Web pages
  - ...
- In the following we will look into
  - how to represent documents
  - how to write a basic search engine over documents

Representing Documents in Python

- Let's write a simple class for text documents.
- How to represent a document in python?
  - What pieces of information do we want to store?

Representing Documents in Python

- How to represent a document in python?
  - What pieces of information do we want to store?
    - **The raw text (string) of the document**
    - **The tokenized text (list of strings)**
    - **The token frequencies of the documents**
    - **A unique identifier for each document**
    - **...**

Token frequencies

- How often did a particular word occur in a text?

id:doc1

---

text:

The raw text string of the document The tokenized text list of strings The token frequencies of the documents A unique identifier for each document

Token frequencies

- How often did a particular word occur in a text?

id:doc1

---

text:
The raw text string of the document The tokenized text list of strings The token frequencies of the documents A unique identifier for each document

'the': 5
'of': 3
'text', 2
'document', 2
'for', 1
...

Token frequencies

- How often did a particular word occur in a text?

id:doc1

---

text:
The raw text string of the document The tokenized text list of strings The token frequencies of the documents A unique identifier for each document

'the': 5
'of': 3
'text', 2
'document', 2
'for', 1
...

- This is an important summary information - we can measure similarity between documents by computing the *"overlap"* of their token frequency tables. (tfidf+cosine similarity)

A simple document class

```
from nltk import FreqDist, word_tokenize
class TextDocument:
    def __init__(self, text, identifier=None):
        """ Tokenizes a text and creates a document."""
        # Store original version of text.
        self.text = text
        # Create dictionaries that maps tokenized,
        # lowercase words to their counts in the document.
        self.token_counts = # TODO
        self.id = identifier
```

- How to tokenize a Text?
- How to create a dictionary from words to counts?

A simple document class

- How to tokenize a Text?
  - Split using regular expressions, e.g.:

```
>>> input = "Dr. Strangelove is the U.S. President's advisor."
>>> re.split(r'\W+', input)
['Dr', 'Strangelove', 'is', 'the', 'U', 'S', 'President', \
 's', 'advisor', '']
```

  - Use nltk:

```
>>> from nltk import word_tokenize
>>> word_tokenize(input)
['Dr.', 'Strangelove', 'is', 'the', 'U.S.', 'President', \
 "'s", 'advisor', '.']
```

- Define a helper function:

```
def normalized_tokens(text):
""" Returns lower-cased tokens.
    >>> normalized_tokens(input)
    ['dr.', 'strangelove', 'is', 'the', 'u.s.', 'president',
     "'s", 'advisor', '.']"""
    pass # TODO
```

A simple document class

How to create a dictionary from words to counts?
⇒ White board.

- Using dictionary comprehension?
- Using a for loop?
- Using the nltk *frequency distribution* (FreqDist)?
  ⇒ check the documentation.

How to create a document

- Document can be created from different starting points ...
  - By setting text and id as strings.
  - By reading plain text file.
  - By reading compressed text file.
  - By parsing XML.
  - By requesting and parsing an HTML file.
  - ...
- However, only one constructor is possible in python.
  ⇒ Arguments of the constructor: the basic elements which are common to all creation scenarios, and define the object (in our case text and document id)
- Similar to multiple constructors:
  Several different static **class methods**, that call the underlying base constructor.
- (This is a simple version of the so-called **factory pattern**)

Multiple static “constructors”

```
class TextDocument:
    def __init__(self, text, identifier=None):
        ...
    @classmethod
    def from_text_file(cls, filename):
        filename = os.path.abspath(filename)
        # TODO: read content of file into string
        # variable ’text’.
        # ...
        return cls(text, filename)

    @classmethod
    def from_http(cls, url, timeout_ms=100):
        ...
```

Class methods

- The first argument (often named `cls`) of a function with the `@classmethod` **function decorator**, refers to the **class itself** (rather than the object).
- The constructor (or any other class method) can then be called from within that function using `cls(...)`
- What is the advantage of using...

```
@classmethod
def from_text_file(cls, filename):
    #...
    return cls(text, filename)
```

- ... over using?

```
@classmethod
def from_text_file(cls, filename):
    #...
    return TextDocument(text, filename)
```

Brainstorming

- What are cases where it can make sense to use factory constructors (i.e. create instances using a method with the `@classmethod` decorator)?

Use cases for Factory Constructors

If you create instances ...

- ... by reading from different sources.
  Examples: files, http, sql-database, mongodb, elastic Search index
- ... by reading from different formats.
  Examples: xml, json, html
- ... by parsing string options.
  Example:

  ```
  a=MyTarClass(extract=True, verbose=True, gzip=True, \
      use_archive_file=True)
  b=MyTarClass.fromOptions("xzvf")
  ```

  (Can you guess what this class might do?)
- ... where the same argument type is interpreted/parsed differently
  Example:

  ```
  a=MyTime.fromTIMEX2("2017-08-01")
  b=MyTime.fromGerman("1. August 2017")
  ```
- ...

Next time: How to write the simple Search Engine

- Demo
- Questions?

Testing with the unittest module

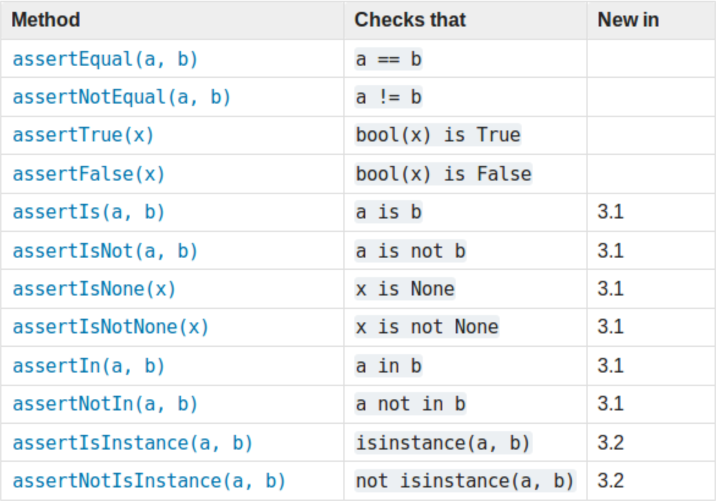
Test-Driven Development (TDD): Recap

- Write tests first (, implement functionality later)
- Add to each test an empty implementation of the function (use the `pass`-statement)
- The tests initially all fail
- Then implement, one by one, the desired functionality
- Advantages:
  - Define in advance what the expected input and outputs are
  - Also think about important boundary cases (e.g. empty strings, empty sets, `float(inf)`, 0, unexpected inputs, negative numbers)
  - Gives you a measure of progress (*"65% of the functionality is implemented"*) - this can be very motivating and useful!

The `unittest` module

- Similar to Java's *JUnit* framework.
- Most obvious difference to `doctest`: test cases are not defined inside of the module which has to be tested, but in a separate module just for testing.
- In that module ...
  - `import unittest`
  - import the functionality you want to test
  - define a class that inherits from `unittest.TestCase`
    - ★ This class can be arbitrarily named, but `XyzTest` is standard, where `Xyz` is the name of the module to test.
    - ★ In `XyzTest`, write member functions that start with the prefix `test...`
    - ★ These member functions are automatically detected by the framework as tests.
    - ★ The tests functions contain `assert`-statements
    - ★ Use the `assert`-functions that are inherited from `unittest.TestCase` (do not use the Python built-in `assert` here)

Different types of asserts



Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Question: … what is the difference between “`a == b`” and “`a is b`”?

Example: using unittest

- test_square.py

```
import unittest
from example_module import square

class SquareTest(unittest.TestCase):
    def testCalculation(self):
        self.assertEqual(square(0), 0)
        self.assertEqual(square(-1), 1)
        self.assertEqual(square(2), 4)
```

Example: running the tests initially

- test_square.py

```
$ python3 -m unittest -v test_square.py
testCalculation (test_square.SquareTest) ... FAIL

======================================================================
FAIL: testCalculation (test_square.SquareTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/ben/tmp/test_square.py", line 6, in testCalculation
    self.assertEqual(square(0), 0)
AssertionError: None != 0

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
$
```

Example: running the tests with implemented functionality

```
$ python3 -m unittest -v test_square.py
testCalculation (test_square.SquareTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
$
```

SetUp and Teardown

- `setUp` and `teardown` are recognized and exectuted automatically before (after) the unit test are run (if they are implemented).
- `setUp`: Establish pre-conditions that hold for several tests.
  Examples:
  - Prepare inputs and outputs
  - Establish network connection
  - Read in data from file
- `tearDown` (less frequently used): Code that must be executed after tests finished
  Example: Close network connection

Example using setUp and tearDown

```
class SquareTest(unittest.TestCase):
    def setUp(self):
        self.inputs_outputs = [(0,0),(-1,1),(2,4)]

    def testCalculation(self):
        for i,o in self.inputs_outputs:
            self.assertEqual(square(i),o)

    def tearDown(self):
        # Just as an example.
        self.inputs_outputs = None
```

Conclusion

- Test-driven development
- Using `unittest` module
- Also have a look at the online documentation!
  `https://docs.python.org/3/library/unittest.html`
- Questions?