Introduction to programming
Lecture 6

UNIVERSITY OF GOTHENBURG

Richard Johansson

October 6, 2015

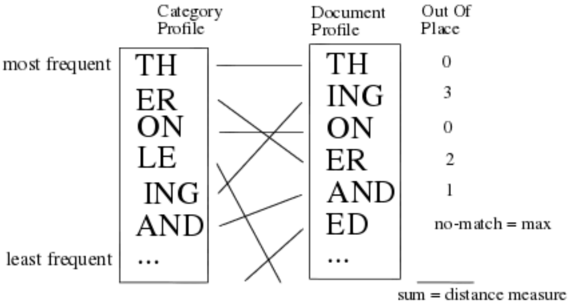
overview of today's lecture

- some details about the 3rd assignment
- two useful variants of dictionaries
- basic theory about algorithm complexity
- more object-oriented programming
- defining your own data structures – if we have time

assignment 3

- in assignment 3 you will work on the problem of **language identification**
  - given some text, guess which language it has
- we solve this problem by creating **language profiles** for a set of languages, to which the unknown texts are compared
- a profile is a frequency table of letter combinations, which we call *n*-grams
  - the *n* refers to the number of letters
- for example, *ing* is a frequent 3-gram in English, so if it is also frequent in the text, then the text is likely in English

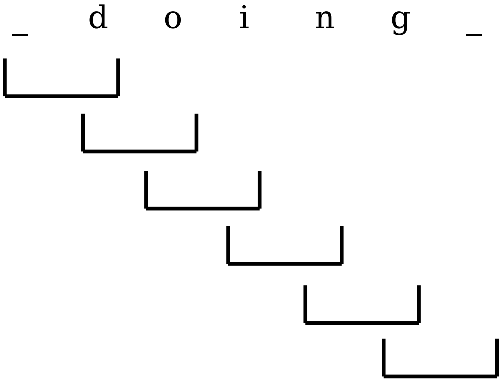
comparing a document profile to a language profile



collecting *n*-gram statistics

```
def collect_ngram_statistics(words, dictionary, n):
    pad = ' '*(n-1)
    for word in words:
        padded_word = '%s%s%s' % (pad,word,pad)
        index = 0
        while index+n <= len(padded_word):
            ngram = padded_word[index:index+n]
            if ngram in dictionary:
                dictionary[ngram] += 1
            else:
                dictionary[ngram] = 1
            index += 1
```

example: collecting bigrams



running the program

- when you execute a Python file (from the editor or from the command line), that file is called the main program
- you can test if a Python file is executed as a main program:

```
if __name__ == '__main__':
    print("This Python file is run as a main program.")
```

UNIVERSITY OF GOTHENBURG

command-line arguments

- when running a program, you can get its **command-line arguments** from the list `sys.argv`:

```
import sys
if __name__ == '__main__':
    inputfile = sys.argv[1]
    outputfile = sys.argv[2]
    ... do something with inputfile and outputfile ...
```

- for instance, assuming that this code is in `dosomething.py`:

```
python dosomething.py input.txt output.txt
```

- . . . then the variable `inputfile` will be `"input.txt"` and `outputfile` will be `"output.txt"`

data persistence

- when our program has carried out some work, we might want to **save** it so that we can reuse it later
- we have already seen how to write to a text file:

```
with open('output.txt', 'w') as f:
    print('this is the output to the text file', file=f)
```

- **pickling** (in other languages called **serializing**): converting a Python object to raw data (a string) so that it can be written to a file and later reloaded
- we can save our data without having to define a file format

```
import pickle
with open('output.data', 'wb') as f:
    pickle.dump(some_object, f)

...
with open('output.data', 'rb') as f:
    reloaded = pickle.load(f)
```

example: saving and loading a frequency table

```
import nltk
import sys
import pickle

def compute_frequencies(filename):
   ...
   return frequencies

if __name__ == '__main__':
    inputfile = sys.argv[1]
    outputfile = sys.argv[2]
    freqs = compute_frequencies(inputfile)
    with open(outputfile, 'wb') as f:
        pickle.dump(freqs, f)
```

```
import sys
import pickle

if __name__ == '__main__':
    inputfile = sys.argv[1]
    testword = sys.argv[2]
    with open(inputfile, 'rb') as f:
        freqs = pickle.load(f)
    print(freqs[testword])
```

two useful types of dictionaries

- `defaultdict` and `Counter`
- they live in the `collections` module in the standard library
- they behave just like normal dictionaries, but have some additional advantages to make them more practical
- `https://docs.python.org/3/library/collections.html`

recap: making a frequency table

```
document = ["this", "is", "a", "collection", "of",
            "words", "and", "it", "is", "extracted",
            "from", "a", "text"]

freqs = {}
for word in document:
    if word in freqs:
        freqs[word] += 1
    else:
        freqs[word] = 1

print(freqs)
```

using a defaultdict

- a **defaultdict** is like a normal dictionary, but it will make a new value for keys it hasn't seen before

```
from collections import defaultdict

document = ["this", "is", "a", "collection", "of",
            "words", "and", "it", "is", "extracted",
            "from", "a", "text"]

freqs = defaultdict(int)
for word in document:
    freqs[word] += 1
```

using a Counter

- a **Counter** (note the capital C) is a dictionary specialized for frequency counting

```
from collections import Counter

document = ["this", "is", "a", "collection", "of",
            "words", "and", "it", "is", "extracted",
            "from", "a", "text"]

# almost like before
freqs = Counter()
for word in document:
    freqs[word] += 1

# to get the most frequent:
print(freqs.most_common(3))
```

using a Counter (even simpler)

```
from collections import Counter
document = ["this", "is", "a", "collection", "of",
            "words", "and", "it", "is", "extracted",
            "from", "a", "text"]

freqs = Counter(document)
```

example: counting the part-of-speech tags for each word

The	DT
last	JJ
thing	NN
they	PRP
needed	VBD
was	VBD
another	DT
drag-down	JJ
blow	NN
.	.
,,	,,
That	DT
measure	NN
could	MD
compel	VB
...	...

example: counting the part-of-speech tags for each word (code)

```
from collections import defaultdict, Counter

stats = defaultdict(Counter)

with open('tagged_corpus.txt') as f:
    for l in f:
        word, tag = l.split()
        stats[word][tag] += 1

print(stats['measure'])
```

overview

introduction to the theory of algorithms

more object-oriented programming

making your own data structures

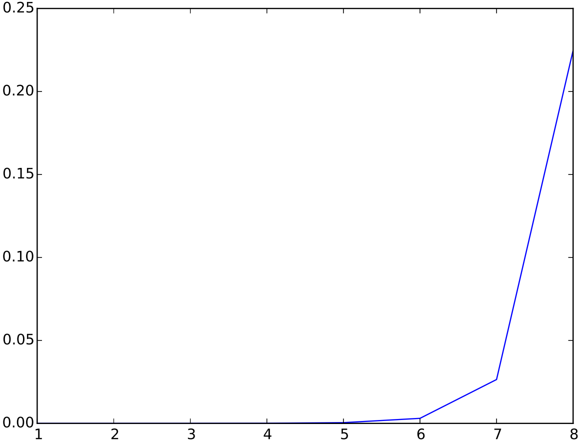
complexity of algorithms

- apart from being correct, we prefer that our program
  - does its job in reasonable **time**
  - doesn't fill the whole **memory** of the machine
- in general, the time and the memory consumption of an algorithm depend on the size of the input
- we will have a look at the **time complexity**: the relation between input size and time
  - conversely, there is a notion of **memory complexity**
- we'll start by comparing a few different sorting algorithms

the "idiot sort algorithm"

1. are the elements sorted yet?
2. if not, shuffle the elements randomly and check again

measuring the time of the idiot sort algorithm



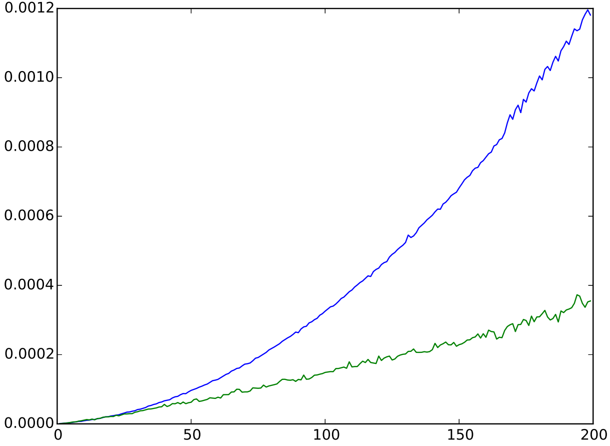
the selection sort algorithm

1. put the lowest element at the first position
2. put the second lowest element at the second position
3. . . .

the quicksort algorithm

- if the list is empty or has just one element, we're done
- else
  1. select a pivot element p
  2. quicksort the sublist L of elements less than p
  3. quicksort the sublist G of elements greater than p
  4. arrange the elements in the order L, p, G

execution time of selection sort (blue) and quicksort (green)



notation for time complexity

- we use the **ordo notation** for time complexity
- for instance, locating an element in a list (`if x in lst`) of length N has a time complexity of $O(N)$
- this means that the relation between time and the size N is something like

$$\text{time} = \text{something} \cdot N + \text{something}$$

  where *something* is a constant number that depends on your machine, the programming language, etc
- we can distinguish **average-case** and **worst-case** complexity

some common terminology

- an algorithm with a time complexity of $O(N)$ is said to run in **linear time**
- $O(1)$: **constant time**
- $O(N^2)$: **quadratic time**
- $O(N^3)$: **cubic time**
- $O(\log N)$: **logarithmic time**
- $O(\exp N)$: **exponential time**

reasoning about time complexity

- when determining the time complexity, we try to reason about how many steps the algorithm will take, depending on the input size N
- in general
  - a single loop over the whole input gives $O(N)$
    - (assuming each step takes constant time!)
  - a loop inside a loop gives $O(N) \cdot O(N) = O(N^2)$
    - or equivalently, calling an $O(N)$ function inside a loop
  - **but** one loop after another gives $O(N) + O(N) = O(N)$

examples

- what is the time complexity of the function `max` in Python?

examples

- what is the time complexity of the function `max` in Python?
  - it just goes through all the elements once, so **linear** ($O(N)$)
  - (assuming that we don't use any complicated key function)

examples

- what is the time complexity of the function `max` in Python?
  - it just goes through all the elements once, so **linear** ($O(N)$)
  - (assuming that we don't use any complicated key function)
- what is the time complexity of selection sort?

examples

- what is the time complexity of the function `max` in Python?
  - it just goes through all the elements once, so **linear** ($O(N)$)
  - (assuming that we don't use any complicated key function)
- what is the time complexity of selection sort?
  - it has a loop inside a loop, so **quadratic** ($O(N^2)$)

data structures

- we use **data structures** to store the data that our program processes
  - lists, sets, dictionaries, . . .
- the selection of a data structure is a tradeoff
  - list: we remember order; fast access; quite fast to add elements at the end but slow elsewhere; slow to test membership
  - set: we don't remember order; fast to add elements; fast to test membership
  - dictionary: key–value mapping; we don't remember insertion order; fast lookup by key; slow lookup by value
- in some cases, we may need to develop our own data structures
  - see last part of this lecture

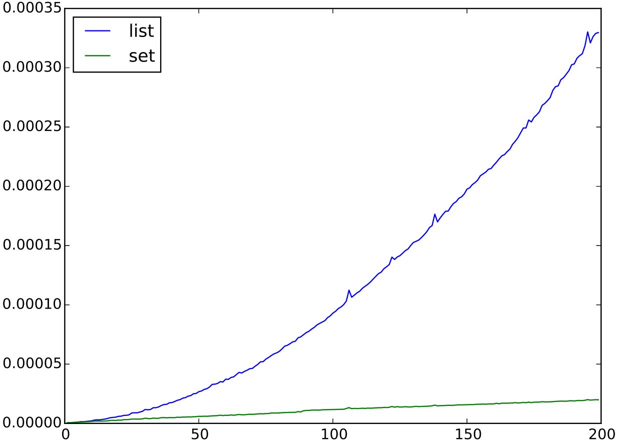
time complexity of our common data structures

- see `https://wiki.python.org/moin/TimeComplexity`
- **list**:
  - accessing an item (`lst[i]`) takes **constant time**
  - append takes **constant time** (practically)
  - `insert`/`delete` take **linear time** (especially near beginning)
  - membership test (`if x in lst`) takes **linear time**
- **dictionary**:
  - lookup (`d[key]`) takes **constant time** (practically)
  - insertion (`d[key] = value`) takes **constant time** (practically)
  - key membership test takes **constant time** (practically)
  - value membership test takes **linear time**
- **set**:
  - add takes **constant time** (practically)
  - membership test takes **constant time** (practically)
- **sorting** in Python has a time complexity of $O(N \cdot \log N)$

example: counting the number of unique elements in a list

```
def count_unique(lst):
    seen_before = []
    for x in lst:
        if x not in seen_before:
            seen_before.append(x)
    return len(seen_before)
```

counting unique elements



memory complexity, briefly

- we want to count the number of tokens in a corpus
- two implementations:
  1. load the corpus into memory, count all the tokens
  2. read the corpus line by line, add the number of tokens on each line to a counter
- which implementation is most efficient in terms of memory?

memory complexity, briefly

- we want to count the number of tokens in a corpus
- two implementations:
  1. load the corpus into memory, count all the tokens
  2. read the corpus line by line, add the number of tokens on each line to a counter
- which implementation is most efficient in terms of memory?
  1. memory complexity linear in the corpus size
  2. memory complexity linear in the maximum line length

overview

introduction to the theory of algorithms

more object-oriented programming

making your own data structures

recap: declaring a class

The name of the class

The name of its superclass

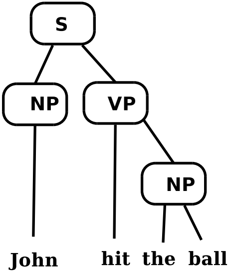
```
class MyClassName (object):

    def __init__(self, inputs):
            ...
            (do something)
            ...

    def some_method(self, inputs):
            ...
            (do something)
            ...
    def some_other_method(self, inputs):
            ...
            return something
```

Constructor definition

Method definitions

example: phrase structure trees

- a **phrase structure tree** is a tree commonly used to represent syntactic structure
- it consists of **phrases** and **words**
- a phrase consists of a phrase label (e.g. NP, VP, . . .) and a list of children (words or other phrases)



UNIVERSITY OF GOTHENBURG

example: phrase structure trees

```
w1 = Word("John")
w2 = Word("hit")
w3 = Word("the")
w4 = Word("ball")

p1 = Phrase("NP", [w1])
p2 = Phrase("NP", [w3, w4])
p3 = Phrase("VP", [w2, p2])
p4 = Phrase("S", [p1, p3])

p4.printout()

print(w4.depth())
```

```
S:
    NP:
        John
    VP:
        hit
        NP:
            the
            ball

3
```

phrase structure trees: the code

```
class Word(object):
    def __init__(self, word):
        self.word = word
        self.parent = None

    def printout(self, ind):
        print(" "*ind + self.word)

    def depth(self):
        if not self.parent:
            return 0
        else:
            return 1 + self.parent.depth()
```

```
class Phrase(object):
    def __init__(self, label, children):
        self.parent = None
        for c in children:
            c.parent = self
        self.children = children
        self.label = label

    def printout(self, ind = 0):
        print(" "*ind + self.label+":")
        for c in self.children:
            c.printout(ind + 4)

    def depth(self):
        if not self.parent:
            return 0
        else:
            return 1 + self.parent.depth()
```

inheritance

- we can say that two classes share some methods by declaring them as derived from a common more general **superclass**
  - some methods are shared between the classes, others are not
- this is how we write:

```
class SomeSubClass(SomeSuperClass):
    ...
```

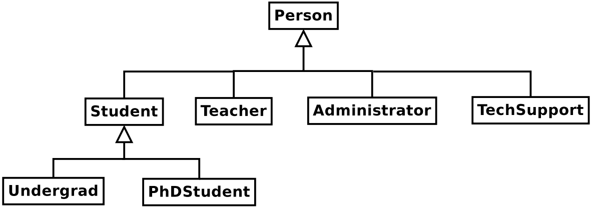
- we say that the subclass **inherits** methods from the superclass

example of design with inheritance

- a hierarchy of inheritance can be quite deep:

```
class Person(object):
    ...
class Student(Person):
    ...
class Teacher(Person):
    ...
class Undergrad(Student):
    ...
```

- note: if `isinstance(x, Undergrad)`, then we also have `isinstance(x, Person)` and `isinstance(x, Student)`



example of design with inheritance

```
class Person(object):
    def __init__(self, name, pnr):
        ...

class Student(Person):
    ...
    def register_at_course(self, course):
        ...

class PhDStudent(Student):
    ...
    def add_publication(self, article):
        ...
```

phrase structure trees with inheritance

```
class Node(object):
    def depth(self):
        if not self.parent:
            return 0
        else:
            return 1 + self.parent.depth()

class Word(Node):
    def __init__(self, word):
        self.word = word
        self.parent = None

    def printout(self, ind):
        print(" "*ind + self.word)
```

```
class Phrase(Node):
    def __init__(self, label, children):
        self.parent = None
        for c in children:
            c.parent = self
        self.children = children
        self.label = label

    def printout(self, ind = 0):
        print(" "*ind + self.label+":")
        for c in self.children:
            c.printout(ind + 4)
```

overview

introduction to the theory of algorithms

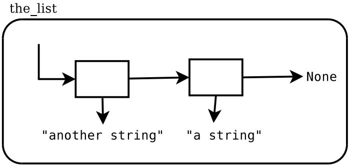
more object-oriented programming

making your own data structures

developing our own data structures: a linked list

- a **linked list** is a data structure consisting of a chain of links, where each link contains a piece of data
- advantages compared to a normal Python list: easy and fast to insert, especially at the start of the list
- disadvantages: complicated and slow to get the *n*-th item

```
class LinkedList(object):
    ...

the_list = LinkedList()
the_list.add_first("a string")
the_list.add_first("another string")
print(the_list.get_first())
```



linked list implementation

```
class Link(object):
    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedList(object):
    def __init__(self):
        self.first = None
    def add_first(self, data):
        self.first = Link(data, self.first)
    def get_first(self):
        if self.first:
            return self.first.data
    def remove_first(self, data):
        if self.first:
            self.first = self.first.next

the_list = LinkedList()
the_list.add_first("a string")
the_list.add_first("another string")
print(the_list.get_first())
```

iterators and iterables

- an **iterator** is an object that has a method called `__next__`
  - `__next__` will generate a new item each time it is called
- an **iterable** is an object that has a method called `__iter__` that will return an iterator
- if an object is iterable, then we can use it in a `for`
  - all data structures such as lists, sets, dictionaries are iterable
  - if we are going through a list, the iterator will remember the position where we are currently looking

```
for x in some_iterable:
    ... do something with x ...
```

example

```
class NumberGenerator(object):
    def __init__(self):
        self.current = 0
    def __next__(self):
        self.current += 1
        if self.current > 10:
            raise StopIteration
        return self.current

class NumberSequence(object):
    def __iter__(self):
        return NumberGenerator()

numbers = NumberSequence()
for n in numbers:
    print(n)
```

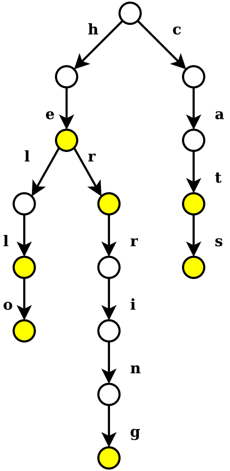
making the linked list iterable

```
class Link(object):
    def __init__(self, data, next):
        self.data = data
        self.next = next

class LinkedListIterator(object):
    def __init__(self, start):
        self.current = start
    def __next__(self):
        if not self.current:
            raise StopIteration
        else:
            out = self.current.data
            self.current = self.current.next
            return out

class LinkedList(object):
    def __init__(self):
        self.first = None
    ...
    def __iter__(self):
        return LinkedListIterator(self.first)
```

```
the_list = LinkedList()

the_list.add_first("test1")
the_list.add_first("test2")
the_list.add_first("test3")

for x in the_list:
    print(x)

print(list(the_list))
```

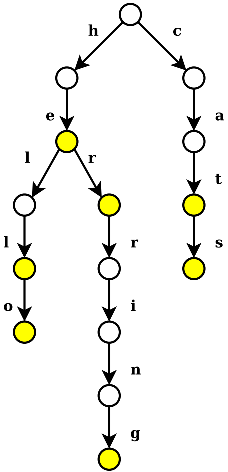
developing our own data structures: letter tree (trie)

- dictionaries are efficient for storing words, but sometimes we need to do more complex things:
  - finding all words starting with *h*, or alphabetically between *abc* and *abx*
  - finding the words most similar to the misspelled word *hering*
  - finding anagrams
  - ...
- a **trie** is a data structure for strings where all strings sharing a prefix are represented as a tree node



trie methods

- to test whether a string `s` is stored in the trie subtree:
  - if `s` is empty, then return `True` if the node is an end node (yellow), `False` otherwise
  - otherwise, if we have a subtree for the first letter `f` of `s`
    - check if the rest of `s` is contained in the subtree
  - otherwise, return `False`
- to insert a string `s` into a trie subtree:
  - if `s` is empty, set the current node as an end node (yellow)
  - otherwise, make sure that there is a subtree for the first letter `f` of `s`
  - . . . then insert the rest of `s` into that subtree



implementing the trie

```
class TrieNode(object):
  def __init__(self):
    self.children = {}
    self.end = False

  def insert(self, s, position):
    if position == len(s):
      self.end = True
    else:
      letter = s[position]
      if not self.children.has_key(letter):
         child = TrieNode()
         self.children[letter] = child
      self.children[letter].insert(s, position+1)

  def contains(self, s, position):
    if position == len(s):
      return self.end
    letter = s[position]
    if not self.children.has_key(letter):
      return False
    return self.children[letter].contains(s, position+1)
```

```
class Trie(object):
  def __init__(self):
    self.root = TrieNode()

  def insert(self, s):
    self.root.insert(s, 0)

  def contains(self, s):
    return self.root.contains(s,0)


t = Trie()

t.insert("hello")
t.insert("he-man")
t.insert("herring")
t.insert("horrible")

print(t.contains("herring"))
print(t.contains("hell"))
```