Python/Scheme for Computational Linguistics

Damir Ćavar
dcavar@unizd.hr

JSSECL 2006

September 2006

Agenda

Agenda

- Introduction

Agenda

- Introduction
- Introduction to Python

Agenda

- Introduction
- Introduction to Python
- Statistics

Agenda

- Introduction
- Introduction to Python
- Statistics
- Clustering

Agenda

- Introduction
- Introduction to Python
- Statistics
- Clustering
- Parsing

Introduction

- Language has formal and static properties, expressed in form of:
  - lexicon
  - rules of a grammar (phonology, morphology, syntax)
  - world knowledge and logical/inferential models (semantics, pragmatics)

Introduction

- Goals:
  - formalization of specifications of *knowledge of language* (e. g. lexicon in databases or XML data structures, phonological and morphological rules in Finite State based grammars, syntax in HPSG or LFG, world knowledge in ontologies or semantic nets)
  - testing the formal models
  - using the models in applications

Introduction

→ computational environment for grammar and logical model development (declarative component) and testing

- databases and data structures for lexical knowledge
- grammar editors and parsers (e. g. Xerox LFG Grammar Writer's Workbench, LKB, PET System)
- data structures for storage of linguistic annotations and analyses

Introduction

- Knowledge of language is not just declarative and/or static:

  → system for application of knowledge of language is procedural
    * How are rules used in language processing?
    * What processing strategy is applied?
    * How does this interact with other linguistic or non-linguistic systems?
    * Grammar changes over time. What are the influential factors? What changes, what does not? How fast does what kind of linguistic property change? What kind of steps can be observed?

Introduction

- Language is:
  - complex
  - processed $\rightarrow$ in time, noisy environments, robustly, ...
  - dynamic: lexicon, grammar, use

- Language also has:
  - quantitative properties

Introduction

- Analysis of language at the quantitative level requires:
  - minimum amount of digitized language data that is quantitatively relevant and significant for various analyses: corpora
  - computational methods for efficient large scale analysis of corpora, and complex statistical, and logical calculations and evaluations

- Creative research in the domain of quantitative and qualitative properties of language, cross-linguistic relations and variations, change, acquisition, processing requires computational aids that are often missing

Introduction

→ Own knowledge of the creation of computational analysis components is crucial!

- You can try to convince somebody to do it for you; you can try to find some existing tool for certain analyses, but you cannot/shouldn't rely on any of these options, but maybe on yourself.
  - You should know how these computations work.
  - You should be able to define and execute them on your own.

Introduction

- Goal of this course:
  - Introduction to computational methods for language analysis.
  - Understanding of formal (programming) languages for this task:
    - Python and/or Scheme

Introduction to Python

- Installing and running Python
- Variables
  - Integers, Floats, Strings, Lists, Tuples, Dictionaries
- Arithmetic Expressions
- Flow control
  - Conditions
  - Loops
  - Functions

Introduction to Python

- Modules
- Classes
- Input and Output
- Exceptions

Statistics

- Counting characters, words
- Creating frequency profiles, maximum likelihood
- N-gram models
- Language Identification
- Calculating Information theoretic measures (Entropy, Mutual Information, Relative Entropy)

Clustering

- K-means document
- Expectation maximization

Parsing

- Parsing a grammar (CFG)
- Simple top-down parsing
- Simple bottom-up parsing
- Chart parsing

Obtaining Python

- Development environment:
  - Python is Free and Open
  - It comes with most systems: FreeBSD, Linux, and Mac OSX
  - It can be installed on any OS, e. g. Microsoft Windows:
    - Python.org
    - ActiveState.com

Readings

- Free online recourses
  - Python.org
  - Dive into Python
  - Thinking in Python
  - A Byte of Python
  - How to think like a computer scientist

Readings

- Books
  - Programming Python [Lutz(1996)]
  - Learning Python [Lutz and Ascher(1999)]
  - Python in a Nutshell [Martelli(2003)]
  - Python Cookbook [Martelli and Ascher(2002)]
  - Python Pocket Reference [Lutz(1998)]

Extensions

- Natural Language Toolkit (NLTK)
- Numerical Python
- SciPy – Scientific tools for Python
- Bob Ippolito's Python Stuff
- Vaults of Parnassus: Python Resources
- Mark Hammond's free stuff

Summary

- **Python V. 2.4.3**
  - High-level open and free programming language
  - System independent
  - Practical relevance (.NET, MONO & WebServices)
  - Rich toolset, NLP toolkits
  - Object oriented, functional, list processing, scripting, Unicode & XML support, low turnaround times
  - GUI: Qt, Tcl/Tk, GTK, Java, Aqua, etc.
  - Integrated in application (e. g. Vim)

Starting Python

- Command line or IDE (or double click on Python script)
- Command line:

```
Damirs:~ dcavar$ python
ActivePython 2.4.3 Build 11 (ActiveState Software Inc.) based on
Python 2.4.3 (#1, Apr  3 2006, 18:07:18)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Command line

- Exit the interactive Python interpreter:
  - Unix: `Ctrl-D`
  - Windows: `Ctrl-Z`
  - Commands:

```
>>> raise SystemExit
```

or

```
>>> import sys
>>> sys.exit()
```

Interaction

- Hello-world example:

```
>>> print "Hello world!"
Hello world!
>>>
```

- `helloworld.py` from within the interactive Python interpreter:

```
>>> execfile("helloworld.py")
Hello world!
>>>
```

Interaction

- Via command-line and file: `helloworld.py`

```
Damirs:~ dcavar$ python helloworld.py
Hello world!
Damirs:~ dcavar$
```

- and remaining in interactive mode after execution:

```
Damirs:~ dcavar$ python -i helloworld.py
Hello world!
>>>
```

Calculating with Python

```
>>> 5 + 4
9
>>> 5 * 3
15
>>> 6 / 2
3
>>> 7 - 3
4
>>> (4 - 2) * 5
10
>>> 4 - 2 * 5
-6
```

Variables

- Dynamically typed
  - Types do not have to be declared in the program.
  - Types of variables can change during program flow, i. e. integers can become strings or lists and vice versa.
- Garbage collection
  - No allocation and memory handling for variables and their content from the programmers perspective.

Integers

- Example: integers and simple arithmetic

```
>>> myValue = 9
>>> newValue = myValue - 4
>>> myValue
9
>>> newValue
5
>>>
```

Floating point numbers

- Example: floats and integers and simple arithmetic

```
>>> myValue = 9.0
>>> newValue = myValue + 4
>>> myValue
9.0
>>> newValue
13.0
>>>
```

Numeric Operations

Operation	Result
`x + y`	sum of `x` and `y`
`x - y`	difference of `x` and `y`
`x * y`	product of `x` and `y`
`x / y`	quotient of `x` and `y`
`x % y`	remainder of `x / y`
`-x`	`x` negated
`+x`	`x` unchanged
`abs(x)`	absolute value or magnitude of `x`
`int(x)`	`x` converted to integer
`long(x)`	`x` converted to long integer
`float(x)`	`x` converted to floating point
`complex(re,im)`	a complex number with real part `re`, imaginary part `im`. `im` defaults to zero.
`c.conjugate()`	conjugate of the complex number `c`
`divmod(x, y)`	the pair `(x / y, x % y)`
`pow(x, y)`	`x` to the power `y`
`x ** y`	`x` to the power `y`

Strings

- Quoting strings:

```
"This is an example."
'This is an example.'
```

- Escape character for quotes in string:

```
"John said: \"Hello.\""
```

- or simply different quotes:

```
'John said: "Hello."'
```

String Variables

```
>>> text = "Hello world!"
>>> text
'Hello world!'
>>>
```

- `text` is a placeholder for, or name of the string `"Hello world!"`
- `text` refers or points to the string `"Hello world!"`, which is automatically allocated and stored in memory, and freed after no longer in use.

String Operations

- Concatenation:

```
>>> text = "Hello world!"
>>> text = text + " How are you?"
>>> text
'Hello world! How are you?'
>>> other = " OK!"
>>> text = text + other
>>> text
'Hello world! How are you? OK!'
```

String Operations

- Multiplication:

```
>>> text = 2 * text
>>> text
'Hello world! How are you? OK!Hello world! How are you? OK!'
>>> text = "Hello world!"
>>> text = 5 * " " + text + 5 * " "
>>> text
'     Hello world!     '
```

String Operations

- Accessing characters by position:

```
>>> text = "Hello world!"
>>> text[0]
'H'
>>> text[1]
'e'
>>> text[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

String Operations

- Accessing characters by position backwards:

```
>>> text[-1]
'!'
>>> text[-2]
'd'
>>> text[-13]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

String Operations

- Accessing characters by position backwards:

```
>>> text[-1]
'!'
>>> text[-2]
'd'
>>> text[-13]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

String Operations

- Slicing:

```
>>> text[0:3]
'Hel'
>>> text[0:1]
'H'
>>> text[:-1]
'Hello world'
>>> text[1:]
'ello world!'
>>> text[:]
'Hello world!'
```

String Operations

- Assigning to indexed or sliced position:

```
>>> text[1] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> text[1:2] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

String Operations

- Setting indexed or sliced position:

```
>>> text = text[0] + "a" + text[2:]
>>> text
'Hallo world!'
>>> text = text[:1] + "e" + text[2:]
>>> text
'Hello world!'
>>> text = text[:2] + text[3:]
>>> text
'Helo world!'
```

String Operations

- Notes:
  - Forward indexing starts with `0`
  - Backward indexing starts with `-1`
  - `Index out of range` exception occurs if index out of bounds
  - `text` is equivalent to `text[:]`
  - Assignment of values to indexed positions or slices is not possible with `string` types, i. e. strings are *immutable* objects.
  - Changing strings implies internal reallocation of a new string variable, thus expensive memory operations.

String Operations

- Performance issues with string concatenation:
  - In potentially long loops with concatenation operations, instead of:

    ```
    text = text[:1] + "e" + text[2:]
    ```

  - use:

    ```
    text = "".join([text[:1], "e", text[2:]])
    ```

String Operations

- Integers or floats to strings:

```
>>> a = 0.9
>>> str(a)
'0.9'
>>> b = 5
>>> str(b)
'5'
>>> text = text + " " + str(a) + " " + str(b)
>>> text
'Halo world! 0.9 5'
```

String Operations

- Integers or floats to strings:

```
>>> text = text + a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'float' objects
>>> repr(a)
'0.90000000000000002'
>>> repr(b)
'5'
```

String Types

- Escape sequences in strings:
  - Newline (`"\n"`) raw and interpreted:

```
>>> text = "Line 1\nLine 2"
>>> print text
Line 1
Line 2
>>> text = r"Line 1\nLine 2"
>>> print text
Line 1\nLine 2
```

String Types

- Unicode strings:
  - Default: all strings are based on (8–bit) 128 ASCII encoded characters, to change the default, start Python with the option –U:

    `Damirs:~dcavar$ python –U`
  - Prepend Unicode strings with:
    * escape sequences interpreted: `u"text"`
    * raw unicode strings: `ur"text"`
  - Specific encoding: `u"text".encode('utf-8')`
  - Convert from one encoding to another:

    `unicode(text, 'utf-8')`

String Operations

- Strings are *sequence* types:
  - sequences of characters (single byte or multi-byte characters)
  - all sequence types can be subject of sequence operations
    - indexing & slicing
    - membership
    - concatenation & shallow multiplication
    - length
    - min & max value

Sequence Operations

Operation	Result
`x in s`	`True` if an item of `s` is equal to `x`, else `False`
`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
`s + t`	the concatenation of `s` and `t`
`s * n , n * s`	`n` shallow copies of `s` concatenated
`s[i]`	`i`'th item of `s`, origin 0
`s[i:j]`	slice of `s` from `i` to `j`
`s[i:j:k]`	slice of `s` from `i` to `j` with step `k`
`len(s)`	length of `s`
`min(s)`	smallest item of `s`
`max(s)`	largest item of `s`

Sequence Methods

- Some selection:

```
capitalize()
find(sub[, start[, end]]), rfind(sub [,start [,end]])
index(sub[, start[, end]]), rindex(sub[, start[, end]])
lower(), upper()
strip([chars]), lstrip([chars]), rstrip([chars])
replace(old, new[, count])
split([sep [,maxsplit]])
startswith(prefix[, start[, end]]) endswith(suffix[, start[, end]])

>>> text.split()
['Line', '1\nLine', '2']
```

Lists

- Mutable objects
- Sequence types, with any data type in any combination as elements:

```
>>> text.split()
['Line', '1\nLine', '2']
>>> e = [ "test", 56, 6.0, [ "probe", 6 ], 7 ]
>>> e
['test', 56, 6.0, ['probe', 6], 7]
>>> len(e)
5
```

Lists

- Index and slice access:
  - index returns an element
  - slice returns a list

```
>>> e
['test', 56, 6.0, ['probe', 6], 7]
>>> e[0]
'test'
>>> e[1:2]
[56]
>>> e[0:2]
['test', 56]
```

Lists

- Lists are mutable:
  - index or slice access to change elements is possible

```
>>> e
['test', 56, 6.0, ['probe', 6], 7]
>>> e[3] = 45
>>> e
['test', 56, 6.0, 45, 7]
>>> e[2:4] = [ 3, 5 ]
>>> e
['test', 56, 3, 5, 7]
```

Lists

- Care with variable names and assignments:
  - assigning a list variable to another variable does not copy the list!

```
>>> f = e
>>> f
['test', 56, 3, 5, 7]
>>> f[3] = 0.4
>>> e
['test', 56, 3, 0.40000000000000002, 7]
```

Lists

- Care with variable names and assignments:
  - copy of lists assigned to another variable

```
>>> f = e[:]
>>> f
['test', 56, 3, 0.40000000000000002, 7]
>>> e
['test', 56, 3, 0.40000000000000002, 7]
>>> f[3] = 3
>>> f
['test', 56, 3, 3, 7]
>>> e
['test', 56, 3, 0.40000000000000002, 7]
```

Lists

- Detailed control over cloning objects (e. g. lists):
  - `copy` module: `copy` and `deepcopy`

```
>>> import copy
>>> f = copy.copy(e)      # shallow copy
>>> f = copy.deepcopy(e)  # recursive deep copy
>>>
```

Lists

- Concatenation and multiplication of lists:

```
>>> f
['test', 56, 3, 456, 3, 7]
>>> f = 2 * f
>>> f
['test', 56, 3, 456, 3, 7, 'test', 56, 3, 456, 3, 7]
>>> f = f + [ 34]
>>> f
['test', 56, 3, 456, 3, 7, 'test', 56, 3, 456, 3, 7, 34]
```

List Operations

Operation	Result
`s[i] = x`	item `i` of `s` is replaced by `x`
`s[i:j] = t`	slice of `s` from `i` to `j` is replaced by `t`
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of `t`
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	same as `s[len(s):len(s)] = [x]`
`s.extend(x)`	same as `s[len(s):len(s)] = x`
`s.count(x)`	return number of `i`'s for which `s[i] == x`
`s.index(x[, i[, j]])`	return smallest `k` such that `s[k] == x and i <= k < j`
`s.insert(i, x)`	same as `s[i:i] = [x]`
`s.pop([i])`	same as `x = s[i]; del s[i]; return x`
`s.remove(x)`	same as `del s[s.index(x)]`
`s.reverse()`	reverses the items of `s` in place
`s.sort([cmp[, key[, reverse]]])`	sort the items of `s` in place

Tuples

- Immutable ordered sequences:
  - Usually more efficient than list objects

```
>>> e = ( 1, "test", 7.0, ( 3, 5 ), [ 6, 2, "probe" ] )
>>> e
(1, 'test', 7.0, (3, 5), [6, 2, 'probe'])
>>> e[3]
(3, 5)
>>> e[3:]
((3, 5), [6, 2, 'probe'])
```

Tuples

- Elements in tuples can be mutable:

```
>>> e
(1, 'test', 7.0, (3, 5), [6, 2, 'probe'])
>>> e[4][0] = 5
>>> e
(1, 'test', 7.0, (3, 5), [5, 2, 'probe'])
>>> e[3][1] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
```

Dictionaries

- Data structures for key-value pairs (Hash-tables):
  - Fast access to large data collections based on keys and values.
  - A dictionary is an **unordered** collection of key-value pairs.
  - There can only be **one** key with **one** corresponding value in **one** dictionary!
  - **Valid keys** can only be **immutable objects**!
  - Typical CL application is dictionaries, frequency tables, n-gram models, rule sets, etc.
  - This is one of the most important data structures in the following!

Dictionaries

- Using dictionaries:

```
>>> e = { "key1":"value1", "key2":[ 1, 2 ], "key3":34 }
>>> e
{'key3': 34, 'key2': [1, 2], 'key1': 'value1'}
>>> e["key4"] = 34
>>> e["key2"] = 23
>>> e
{'key3': 34, 'key2': 23, 'key1': 'value1', 'key4': 34}
>>> e["key1"]
'value1'
```

Dictionaries

- Accessing and checking for keys:

```
>>> e.keys()
['key3', 'key2', 'key1', 'key4']
>>> e.has_key("key1")
True
>>> e.has_key("key65")
False
>>> e["key65"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'key65'
```

Dictionaries

- Key and value types:

```
>>> e[1] = 34
>>> e["house"] = "Haus"
>>> e["house"] = [ "N", "Haus" ]
>>> e["house"] = ( "N", "Haus" )
>>> e[ ( 1, 2 ) ] = 87
>>> e[ [ 1, 2 ] ] = 96
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Flow Control

- Conditions
- Loops
- Functions

Conditions

- Conditional execution of code blocks (`True`/`False`, certain values)
  - Indention-based code blocks (either space- or tab-marked)
  - Lines belonging to one code block have the same amount of space- or tab-characters in the beginning of the line.

```
>>> if 1 > 0:
...     print "Hello!"
... else:
...     print "Hallo!"
...
Hello!
```

Conditions

- Testing conditions with: `<`, `>`, `>=`, `<=`, `==`, `!=`, `and`, `or`, `not`

```
if i > 0:
   print "i is positive"
elif i == 0:
   print "i equals 0"
else:
   print "i is negative"

if "a" not in [ "test", "b", "c" ]:
   pass
else:
   print "a"
```

Conditions

- Testing for an element in a sequence:
  `if x in y`
  or
  `if x not in y`
  - `y` can be a string, tuple, list
- Empty code blocks: `pass`

Conditions

- Testing over variable values and content: integers (if value is `0`, return `False`, else return `True`)

```
>>> a = 5
>>> if a:
...     print "test"
...
test
>>> a = 0
>>> if a:
...     print "test"
...
>>>
```

Conditions

- Testing over variable values and content: strings (if string is empty, return `False`, else return `True`)

```
>>> a = "Hello"
>>> if a:
...     print "test"
...
test
>>> a = ""
>>> if a:
...     print "test"
...
>>>
```

Loops

- Looping over values:

```
>>> a = 5
>>> while a > 0:
...     print "a =", a
...     a = a - 1
...
a = 5
a = 4
a = 3
a = 2
a = 1
>>>
```

Loops

- Looping over values with internal break condition:

```
>>> a = 5
>>> while True:
...     print "a =", a
...     a -= 1
...     if a == 0:
...         break
...
a = 5
a = 4
a = 3
a = 2
a = 1
```

Loops over Sequences

- Sequential sequence processing:

```
>>> a = [ "a", "b", "c" ]
>>> for i in a:
...     print i
...
a
b
c
```

Loops over Sequences

- Inefficient sequential sequence processing:

```
>>> a = [ 1, 2, 3 ]
>>> b = []
>>> for i in a:
...     b.append(float(i))
...
>>> b
[1.0, 2.0, 3.0]
```

Loops over Sequences

- More efficient: list comprehension
  - Loop over all list elements, apply a function to each of them, and return a list with the resulting values.
  - This is the fastest an most efficient solution in Python!

```
>>> a = [ 1, 2, 3 ]
>>> b = [ float(i) for i in a ]
>>> b
[1.0, 2.0, 3.0]
```

Loops over Sequences

- Index based loop: `range(n)`
  - Returns as default a list of numbers from `0` till `n-1`
  - Looping over the index positions of a list via `range(len(text))`
  - Necessary to access elements from sequences (lists, tuples, strings) by position

Functions

- Functions and recursion:

```
>>> def fact(num):
...     if num == 1:
...             return 1
...     else:
...             return num * fact(num - 1)
...
>>> fact (3)
6
>>> fact(6)
720
```

Return Values of Functions

- Unpacking of function return values:

```
>>> def convert(text):
...     return text, text.lower(), text.upper()
...
>>> convert("Hello")
('Hello', 'hello', 'HELLO')
>>> a, b, c = convert("Hello")
>>> a
'Hello'
>>> b
'hello'
>>> c
'HELLO'
```

Functions and Modules

- Functions stored in Python code files
  - Reuse of functions via `import` in Python programs
  - Naming conventions! Example: `string`

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import string
>>> dir()
['__builtins__', '__doc__', '__name__', 'string']
>>> dir(string)
['Template', '_TemplateMetaclass', '__builtins__', '__doc__', (...)
'center', 'count', 'digits', 'expandtabs', 'find', 'hexdigits', (...)]
```

Functions and Modules

- Using imported functions: `module-name.function-name`

```
>>> import string
>>> string.split("Hello world!")
['Hello', 'world!']
>>> import math
>>> math.log(2)
0.69314718055994529
>>> math.log(1)
0.0
```

Functions and Modules

- Importing only specific functions:
  `from` module `import` function

```
>>> from math import log
>>> log(2)
0.69314718055994529
>>> log(1)
0.0
```

Input and Output

- Reading data from files: `python readfile.py`

```
file = open("readfile.py")
text = file.read()
file.close()
print text
```

Input and Output

- Reading data from files line by line: `python readfilel.py`

```
file = open("readfilel.py")
text = file.readlines()
file.close()
for i in text:
   print i,
```

Input and Output

- Reading data from files line by line and processing each line immediately:
  `python readfilelp.py`

```
file = open("readfilelp.py", "r")
line = file.readline()
while line:
   print line,
   line = file.readline()
file.close()
```

Input and Output

- Compact reading of data from file: `python readfilec.py`

```
print = open("readfilec.py").read()
```

Input and Output

- Writing data to file: `python writefile.py`

```
text = "This is a test."
file = open("test.txt", "w")
file.write(text)
file.close()
```

Input and Output

- Appending data to a file (creating it, if it doesn't exist):
  `python writefile.py`

```
text = "This is a test."
file = open("test.txt", "a")
file.write(text)
file.close()
```

Input and Output

- Writing Unicode (UTF-8) text data to a file:
  `python writefileHR.py`

```
text = u"Pokušati ćemo pisati hrvatski tekst."
file = open("test.txt", "w")
file.write(text)
file.close()
```

```
Damirs:~/Code dcavar$ python writefileHR.py
sys:1: DeprecationWarning: Non-ASCII character '\xc5' in file writefileHR.py on line
Traceback (most recent call last):
  File "writefileHR.py", line 3, in ?
    file.write(text)
UnicodeEncodeError: 'ascii' codec can't encode characters in position 4-5: ordinal n
```

Input and Output

- Writing Unicode (UTF-8) text data to a file:
  `python writefileHR1.py`

```
# -*- coding: utf8 -*-

import codecs

text = u"Pokušati ćemo pisati hrvatski tekst."
file = codecs.open("test.txt", "w", "utf8")
file.write(text)
file.close()
```

Exceptions

- Various functions throw exceptions:
  `python readfileN.py`

```
file = open("some.txt")
text = file.read()
file.close()
print text

Traceback (most recent call last):
  File "readfileN.py", line 1, in ?
    file = open("some.txt")
IOError: [Errno 2] No such file or directory: 'some.txt'
```

Exceptions

- Various functions throw exceptions:
  `python readfileNE.py`

```
try:
   file = open("some.txt")
   text = file.read()
   file.close()
except IOError:
   print "Cannot open file some.txt."
else:
   print text
```

Comments

- Comments in the code:

```
# reading in the data
#
file = open("some.txt")
# text = file.read()
file.close()
```

Documentation

- Every file, function, or class can be documented:

```
"""
File: test.py
Author: Damir Cavar
Date: 05-09-20
Purpose: Showing Python documentation features.
"""

def test(text):
"""Testing the print features.
   Parameter: text, a string containing the text to be printed."""
print text
```

Documentation

- Generating documentation documents with `pydoc`:
  - Help on `pydoc` on the web and by starting `pydoc` without parameters in the command-line shell:
    `Damirs: / dcavar$ pydoc`

```
Damirs:~/ dcavar$ pydoc -w ./test.py
wrote test.html
```

Classes

- Object oriented encapsulation of data and functions:
  - specific data structures
  - specific methods to manipulate the encapsulated data
  - modularity and reusability, complexity etc.
  - Example:
    - Phrase structure rules of the type: `NP -> DET N`
    - Structure: left-hand side, arrow, right-hand side
    - LHS: only one symbol
    - RHS: any number of symbols
    - Symbols: any combination of non-whitespace characters

Grammar Parsing

- Reading a grammar from a file into a data-structure:
  - opening a file
  - reading in line by line
  - skipping comment lines or empty lines
  - splitting lines with rules into LHS and RHS
  - storing LHS with its corresponding RHS

Grammar Parsing

- Grammar parser:
  - example grammar: `grammar.txt`
  - writing grammar parser...
  - see `grammar.py`

Grammar Parsing

- Conceptual questions:
  - What will be the use of the code?
    - Who will use it how for what purpose?
  - What data structures do we need?
    - Determine all the major storage variables.
  - What shall we be able to do with the data structure?
    - Determine the major functions to process, access, change, use the internal data structures.

Parsing and Phrase Structure Grammar

- Top-down parsing:
  - Replace goal symbol with symbols and symbols with terminals until the terminals match.
- Bottom-up parsing:
  - Replace terminals with symbols and symbols with symbols until the goal symbol is reached.

Parsing

- Parsing strategies:
  - Top-down parsing
  - Bottom-up parsing

- Processing strategies:
  - Breadth first
  - Depth first

Parsing

- What problems do different strategies have?
  - Recursion
  - Multiple choices
    - Backtracking
    - Agenda

Parsing

- Implementation: (`TDAParser.py`)
  - Top-down with weak generative capacity:
    - Input 1: tokenized sentence
    - Input 2: grammar and goal-symbol
    - Output: yes/no or successful/failed parse

Chart Parser Implementation

- Main part:
  - Program initialization vs. module import:

```
if __name__ == "__main__":
   parse(["John", "kissed", "Mary"])
```

Parsing

- Top-down implementation:
  - Input 1: tokenized sentence
  - Input 2: goal-symbol
    - * Assume two lists: Input1 and Input2
    - * Success: replace symbols in Input2 until Input1 equals Input2
    - * Failure: no replacement possible, Input1 does not equal Input2

Parsing

- Top-down implementation:
  - see code example in ZIP file TDA1.zip:
    1. TDAParser.py
    2. grammar.txt
    3. grammar.py

Parsing Strategy

- Two lists:
  - Input list: `[ 'John', 'kissed', 'Mary' ]`
  - Parse list: `[ 'S' ]`
- If lists are equal after applying replacement on the Parse list, the parse is successful.

Parsing Strategy

- Reduce lists every time there is a partial match:
  - Input list: `[ 'John', 'kissed', 'Mary' ]` → `[ 'kissed', 'Mary' ]`
  - Parse list: `[ 'John', 'VP' ]` → `[ 'VP' ]`
- Intuition: there is a parse for the sentence if `[ 'kissed', 'Mary' ]` can be derived from `[ 'VP' ]`
- Continue parsing with the reduced lists

Parsing Strategy

- Conditions:
  - Parsing is successful if we end up with:
    * Input list = [ ]
    * Parsing list = [ ]
  - Parsing fails if:
    * One list is empty and the other not
    * Both lists are not empty and there is no possibility to reduce them or apply further replacement

Parsing Strategy

- Improvement of the parsers:
  - Ordering of rules: more common rules first
    - Try manipulating the order of rules in the grammar, e.g. the VP rules with transitive or intransitive VPs
  - Number of symbols in RHS cannot be bigger than number of symbols and/or terminals in the input
  - Tagging the input first
  - Depth-first rather than breadth-first with respect to the agenda

Parsing Strategy

- Improvement of parser:
  - Tagging the input first
  - Depth-first rather than breadth-first with respect to the agenda
  - Recursive function calls vs. loop

Parsing Strategy

- Bottom-up parsing:
  - Replace the input tokens until the input list consists of the goal symbol only.
  - Example implementation: loop and not recursive function call
    - Advantage: no stack-overflow with long input sentences.
  - Example: `BUAParser.py`

Parsing Strategy

- Problems:
  - Dependencies between tokens in the clause
    - agreement, binding, negative polarity and other particles, idioms, anaphoric relations, periphrastic constructions etc.
  - Structures depend on the properties of tokens and vice versa
    - transitivity of verbs, selectional properties

Parsing Strategy

- Problems:
  - Grammars
    - recursion: unlimited number of elements on the agenda?
    - empty elements or traces

Parsing Strategy

- Problems observed:
  - Reanalysis of already analyzed constituents
  - Search through all grammar rules
- Solution:
  - Memorize analyzed constituents
  - Choose appropriate rules

Parsing Strategy

- Solution:
  - Chart Parsing
    - Chart as memory
    - Selection of relevant rules from grammar

Chart Parsing

- Chart:
  - Storage for complete and incomplete constituents
  - Edges
    - Dotted rule
    - Index

Chart Parsing

- Chart:
  - Storage for complete and incomplete constituents
  - Edges
    - Dotted rule: VP $\rightarrow$ V $\bullet$ NP
    - Index:
      - Left and right position of the edge span
      - Position of the dot in the RHS

Chart Parsing

- Edges:
  - Dotted rule: `VP` $\rightarrow$ `V` $\bullet$ `NP`
    How much of the input at which position matches which part of the RHS of the rule?
  - Example:
    - Input: `[ "John", "loves", "Mary" ]`
    - Edge: ((1, 2, 1, V $\rightarrow$ loves $\bullet$))

Chart Parsing

- Edges:
  - Inactive edge: `(1, 2, 1, V` $\rightarrow$ `loves` $\bullet$`)`
    - Complete constituent
  - Active edge: `(1, 2, 1, VP` $\rightarrow$ `V` $\bullet$ `NP)`
    - Incomplete constituent

Chart Parsing

- Adding edges to chart:
  - **Initialization**
    - Bottom-up strategy: For every token add an inactive edge to chart

      `edge(0, 1, 1, N` $\rightarrow$ `John` $\bullet$`)`

      `edge(1, 2, 1, V` $\rightarrow$ `kissed` $\bullet$`)`

      `edge(2, 3, 1, N` $\rightarrow$ `Mary` $\bullet$`)`
  - **Rule invocation:** Matching edges with rules
  - **Fundamental rule:** Matching active and inactive edges on the chart

Chart Parsing

- Initialization:
  - Top-down strategy:
  - For every token add an inactive edge to chart.
  - For every rule with start-symbol in LHS add active edge to chart:
    - $\texttt{edge(0, 1, 0, S} \rightarrow \bullet \texttt{ NP VP)}$

Chart Parsing

- Rule Invocation:
  - Bottom-up strategy:
  - For every inactive edge on chart:
    - Find rules that have its LHS on their left periphery in RHS
    - Create new edges and add to chart.
  - Example:
    - Inactive edge: `edge(0, 1, 1, N → John •)`
    - Rule: `NP → N`
    - New edge: `edge(0, 0, 0, NP → • N)`

Chart Parsing

- Fundamental Rule:
  - Move inactive edge from agenda to chart
  - For inactive edge find edge that expects it
    - `edge(0, 1, 1, NP` $\rightarrow$ `N` $\bullet$`)`
    - `edge(0, 0, 0, S` $\rightarrow$ $\bullet$ `NP VP)`
  - Add resulting edge to agenda:
    - `edge(0, 1, 1, S` $\rightarrow$ `NP` $\bullet$ `VP)`

Chart Parsing

- Bottom-up:

```
1: Initialize agenda
2: Repeat until edges in agenda
   Process first edge on agenda
   If edge inactive:
      move inactive edge to chart
      Function RuleInvocation
   Function FundamentalRule
```

- Result:

  If chart contains over-spanning edges, these represent possible parses of the input.

Chart Parsing

- Process example:
  - Grammar: `grammar.txt`
  - Implementation: `Charty.py`

Chart Parsing

- Step by step:
  - Initialize chart with the next word of the utterance, i. e. create edge with the lexical rule
  - Find rules in the grammar that consume the symbol of the inactive edges on the chart, i. e. extend the chart with edges that have LHS-symbols of inactive edges at the left periphery of their RHS
  - Create new edges by combining active with inactive edges:
    * end-symbol of one is beginning of other
    * expectation symbol of active edge corresponds to LHS of inactive edge

Chart Parsing

- Motivation:
  - Problems with backtracking (our brute-force) parsers:
    - Repetitive parsing of same token(list)s
    - Repetitive parsing of paths that turned out to be unsuccessful
    - Unknown words and partial structures lead to a failure
  - Chart parser (e. g. Earley parser):
    - Avoid parsing of same token(list)s by memorization in chart
    - Memorize parses for partial structures
      - If a spanning analysis is impossible, the chart contains the partial analyses

Chart Parsing

- Motivation:
  - Chart parser (e. g. Earley parser):
    - ∗ Compact representation for ambiguous structures (multiple parses)

Chart Parsing

- Edges:
  - Directed graph: start point, end point, analysis
  - Input: `[ "John", "kissed", "Mary" ]`
  - Final chart:

(0, 1, N, [John •])	(0, 1, NP, [N •])
(1, 2, V, [kissed •])	(2, 3, NP, [N •])
(2, 3, N, [Mary •])	(1, 3, VP, [V NP •])
(0, 3, S, [NP VP •])	

Chart Parsing

- Bottom-up strategy:
  - Initialization (scan, tagging)
    * Add edges with lexical rules for each token (incrementally)
  - Rule invocation (prediction)
  - Fundamental rule (completion)

Chart Parsing

- Bottom-up strategy:
  - Rule Invocation:
    For every inactive edge on chart:
    * Find rules that have its LHS on their left periphery in RHS.
    * Create new edges and add to chart.

Chart Parsing

- Bottom-up rule invocation example:
  - Inactive edge:
    `edge(0, 1,` $N \rightarrow$ `John` $\bullet$`)`
  - Rule:
    `NP` $\rightarrow$ `N`
  - New edge:
    `edge(0, 0, NP` $\rightarrow \bullet$ `N)`

Chart Parsing

- Fundamental Rule:
  - For every active edge find expected inactive edge:
    `edge(0, 1, N → John •)`
    `edge(0, 0, NP → • N)`
  - Merge edges and add resulting edge to chart:
    `edge(0, 1, NP → N •)`

Chart Parsing

- Top-down strategy:
  - Initialization
    * Add edges with rules with goal symbol on LHS (incrementally)
  - Rule invocation (prediction)
  - Fundamental rule (completion)

Chart Parsing

- Top-down strategy:
  - Rule Invocation:
    For every active edge on chart:
    * Find rules that have its left peripheral symbol from the expected RHS on their LHS. The left peripheral symbol from the expected RHS is the first symbol following the DOT.
    * Create new edges and add to chart.

Chart Parsing

- Top-down rule invocation example:
  - Active edge:
    `edge(0, 0,` S $\rightarrow$ • `NP VP)`
  - Rule:
    `NP` $\rightarrow$ `N`
  - New edge:
    `edge(0, 0, NP` $\rightarrow$ • `N)`

Chart Parsing

- Top-down rule invocation depth-first:
  - Active edge:
    `edge(0, 0, S` $\rightarrow$ • `NP VP)`
  - Rules:
    NP $\rightarrow$ N;
    N $\rightarrow$ John
  - New edges:
    `edge(0, 0, NP` $\rightarrow$ • `N)`
    `edge(0, 0, N` $\rightarrow$ • `John)`

Chart Parsing

- Top-down after rule invocation and fundamental rule:
  - New edges:
    edge(0, 1, S $\rightarrow$ NP • VP)
    edge(0, 1, NP $\rightarrow$ N •)
    edge(0, 1, N $\rightarrow$ John •)

Chart Parsing

- Top-down rule invocation breadth-first:
  - Active edge:
    edge(0, 0, S $\rightarrow$ • NP VP)
  - Rules:
    NP $\rightarrow$ N; VP $\rightarrow$ V NP
  - New edges:
    edge(0, 0, NP $\rightarrow$ • N)
    edge(0, 0, VP $\rightarrow$ • V NP)

Chart Parsing

- Fundamental Rule:
  - For every active edge find expected inactive edge:
    `edge(0, 0, NP` $\rightarrow$ • `N)`
    `edge(0, 1, N` $\rightarrow$ `John` •`)`
  - Merge edges and add resulting edge to chart:
    `edge(0, 1, NP` $\rightarrow$ `N` •`)`

Chart Parsing

- Fundamental Rule:
  - For every active edge find expected inactive edge:
    `edge(0, 0, S` $\rightarrow$ • `NP VP)`
    `edge(0, 1, NP` $\rightarrow$ `N` •`)`
  - Merge edges and add resulting edge to chart:
    `edge(0, 1, S` $\rightarrow$ `NP` • `VP)`

Chart Parsing

- Rule Invocation:
  - Dependent of parsing strategy.
- Fundamental Rule:
  - Independent of parsing strategy.

Chart Parsing

- Differences between top-down and bottom-up parsing:
  - TD: Disambiguates by position.
    - *Calls from Alaska are expensive.*
  - BU: Lexically driven.
  - TD: Has to handle recursion.

Chart Parser Implementation

- Necessary components:
  - Chart
  - Initialization
  - Rule Invocation
  - Fundamental Rule
  - Program Flow-Control

Chart Parser Implementation

- Chart:
  - Storage for edges
  - Edges:
    - start point
    - end point
    - rule
    - dot position

Chart Parser Implementation

- Edge:
  - List of elements:
    `edge = [ 0, 1, 1, "N", "John" ]`
    * integer for start point
    * integer for end point
    * integer for dot position
    * string for rule left-hand side
    * string for rule right-hand side

Chart Parser Implementation

- Chart:
  - Storage for edges
  - List of edges:
    - `chart = [ ]` or

```
chart = [ [ 0, 1, 1, "N", "John" ],
          [ 1, 2, 1, "V", "kissed" ],
          [ 2, 3, 1, "N", "Mary" ] ]
```

Chart Parser Implementation

- Define functions:
  - Initialize: `def initialize():`
  - Rule Invocation: `def ruleInvocation():`
  - Fundamental Rule: `def fundamentalRule():`
  - Parsing Loop: `def parse():`

References

[Lutz(1996)] Mark Lutz. *Programming Python*. O'Reilly & Associates, Inc., Bonn; Sebastopol, CA, 2nd, march 2001 edition, 1996. ISBN 0-596-00085-5.

[Lutz(1998)] Mark Lutz. *Python Pocket Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, third edition, february 2005 edition, 1998. ISBN 0-596-00940-2. URL `http://www.oreilly.com/catalog/pythonpr3/`.

[Lutz and Ascher(1999)] Mark Lutz and David Ascher. *Learning Python*. O'Reilly & Associates, Inc., Sebastopol, CA, 2nd, 2004 edition, 1999. ISBN 1-56592-464-9. URL `http://www.oreilly.com/catalog/lpython/`.

[Martelli(2003)] Alex Martelli. *Python in a Nutshell*. O'Reilly & Associates, Inc., March 2003. ISBN 0-596-00188-6.

[Martelli and Ascher(2002)] Alex Martelli and David Ascher, editors. *Python Cookbook*. O'Reilly & Associates, Inc., Sebastopol, CA, 2002. ISBN 0-596-00167-3. URL http://safari.oreilly.com/0596001673;http://www.oreilly.com/catalog/pythoncook.