A quick introduction to Lisp

(by a dummy)

Pierre-Yves Dupont

07/04/2014

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Much faster: why?

- Operations faster in NumPy (*C* libraries)
- Operations on vectors possible

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Much faster: why?

- Operations faster in NumPy (*C* libraries)
- Operations on vectors possible

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Simple python code:

```
N = 1e4
a = numpy.random.random((N))
b = numpy.random.random((N))

c = numpy.zeros((N,N))

for i in range(len(a)):
 for j in range(len(b)):
 c[i] = a[i] + b[j]
```

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Simple python code:

```
N = 1e4
a = numpy.random.random((N))
b = numpy.random.random((N))

c = numpy.zeros((N,N))

for i in range(len(a)):
 for j in range(len(b)):
 c[i] = a[i] + b[j]
```

Execution in $\approx$ 1.23 minutes

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Should be written:

```
N = 1e4
a = numpy.random.random((N))
b = numpy.random.random((N))

c = numpy.zeros((N,N))# not mandatory

c = a + b
```

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Should be written:

```
N = 1e4
a = numpy.random.random((N))
b = numpy.random.random((N))

c = numpy.zeros((N,N))# not mandatory

c = a + b
```

Execution in less than a second!

Remark: Use Numpy vectors (implicit vectorization during operation on vectors)

Conclusion:
You don’t necessarily need Lisp, C++, Java, . . . to build an efficient code

Introduction to Lisp

- Short introduction
- Some myths and facts
- What Lisp offers

Introduction to Lisp

- Short introduction
- Some myths and facts
- What Lisp offers

Introduction to Lisp

- Short introduction
- Some myths and facts
- What Lisp offers

Some history

1958	Lisp, Lisp 1.5 (John McCarthy - AI pope): LISP = A LISt Processing language. One year after Fortran
1962	First Lisp compiler (Maclisp)
1975	Scheme (one of the two main dialects of Lisp)
1984	Common Lisp (1994 for the ANSI version)
2007	Clojure (Lisp over a VM (Python , Java, Ruby) or compiled to JS - used as a macro system)

Some history

1958	Lisp, Lisp 1.5 (John McCarthy - AI pope): LISP = A LISt Processing language. One year after Fortran
1962	First Lisp compiler (Maclisp)
1975	Scheme (one of the two main dialects of Lisp)
1984	Common Lisp (1994 for the ANSI version)
2007	Clojure (Lisp over a VM (Python , Java, Ruby) or compiled to JS - used as a macro system)

Some history

1958 Lisp, Lisp 1.5 (John McCarthy - AI pope): LISP = A LISt Processing language. One year after Fortran

1962 First Lisp compiler (Maclisp)

1975 Scheme (one of the two main dialects of Lisp)

1984 Common Lisp (1994 for the ANSI version)

2007 Clojure (Lisp over a VM (Python , Java, Ruby) or compiled to JS - used as a macro system)

Some history

- 1958 Lisp, Lisp 1.5 (John McCarthy - AI pope): LISP = A LISt Processing language. One year after Fortran
- 1962 First Lisp compiler (Maclisp)
- 1975 Scheme (one of the two main dialects of Lisp)
- 1984 Common Lisp (1994 for the ANSI version)
- 2007 Clojure (Lisp over a VM (Python , Java, Ruby) or compiled to JS - used as a macro system)

Some history

1958	Lisp, Lisp 1.5 (John McCarthy - AI pope): LISP = A LISt Processing language. One year after Fortran
1962	First Lisp compiler (Maclisp)
1975	Scheme (one of the two main dialects of Lisp)
1984	Common Lisp (1994 for the ANSI version)
2007	Clojure (Lisp over a VM (Python , Java, Ruby) or compiled to JS - used as a macro system)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

LISP

- Functional programming object: each expression in LISP is a function that returns a value. Ex: numbers is a function returning itself. . .
- Object layer (CLOS)
- Polish prefix notation: *(+ 1 2)* and not *1 + 2*
- Interpretive language: LISP programs are interpreted by the LISP interpreter (example: *clisp*). Compilers exist (much faster)
- Most of AI programs in US are written in LISP. (PROLOG for Europe)
- Native (and efficient) Tree Data Structure
- Source code as a data structure: flexible macro system
- ASDF (Another System Definition Facility) libraries manager (Gem, CPAN)

Example: "Hello world"

Python

```
print "Hello world" #python 2.x
print("Hello world")#python 3.x
```

Lisp

```
(print "Hello world")
```

Example: Factorial (recursive)

Python

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n-1)
```

Lisp

```
(defun factorial (n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))
   )
)
```

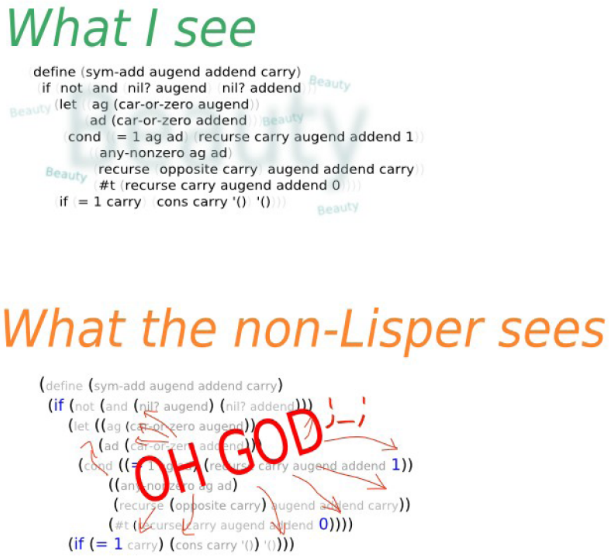
Myths

Myth: LISP = Lost In Stupid Parenthesis?

Myths

Myth: LISP = Lost In Stupid Parenthesis? Fact: Question of point of view. . .

Myths



```
(define (sym-add augend addend carry)
 (if (not (and (nil? augend) (nil? addend)))
   (let ((ag (car-or-zero augend))
         (ad (car-or-zero addend)))
     (cond ((= 1 ag ad) (recurse carry augend addend 1))
           ((any-nonzero ag ad)
            (recurse (opposite carry) augend addend carry))
           (#t (recurse carry augend addend 0))))
   (if (= 1 carry) (cons carry '()) '())))
```

Myths

Myth: There are no libraries in Lisp

Myths

Myth: There are no libraries in Lisp
Fact: About 2000 libraries in QuickLisp

Myths

Myth: There are no libraries in Lisp
Fact: About 2000 libraries in QuickLisp
vs. about 27,500 in PyPI. . .

Myths

Myth: Lisp is slow

Myths

Myth: Lisp is slow
Fact: Fastest dynamic language. Can be faster than Java or C

Myths

Myth: Lisp is slow
Fact: Fastest dynamic language. Can be faster than Java or C
K Nucleotide benchmark: 104 sec for Lisp vs. 923 for Java
benchmarksgame.alioth.debian.org/u32q/lisp.php

Exact computation of fractions

```
[1]> (/ 1 3)
1/3
[2]> (* (/ 1 3) 10)
10/3
[3]> (* (* (/ 1 3) 10) 1/2)
5/3
[4]> (format t "~d" x)
1/3
[5]> (format t "~f" x)
0.33333334
```

Macros

- A macro is a piece of code expanded at compile-time in a resulting form that is compiled by the compiler
- macros do not evaluate their arguments
- nightmare to debug!

Macros

- Code generator
- Tool for abstraction
- Need a compilation step by definition
- Used for:
  - code generation, including objects (AI)
  - creation of new operators (ex: until)
  - parser generation
  - execution of code at compilation time

Creation of an operator using macros

Unless operator:

```
(defmacro unless⋆ (test expr)
`(if, test nil, expr))

(unless⋆ nil(println "This should print"))
(unless⋆ t(println "This shouldn’t print"))
```

(Useless but simple) macro example

Iteration on Prime Numbers:
Utility functions

```
(defun primep (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number)
     never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
\end{lisp
You want to be able to write:
\begin{lisp}
(do-primes (p 0 19)
  (format t "~d " p))
```

(Useless but simple) macro example

Iteration on Prime Numbers:
The macro

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start)
   (next-prime (1+ ,var))))
       ((> ,var ,end))
     ,@body))

(do-primes (p 0 19)
  (format t "~d " p))

2 3 5 7 11 13 17 19
```

Strength of Lisp

- efficient to work on lists, including linked lists and hash maps (sorts, pivots, accession to elements,. . .) because Lisp implementation is based on lists and linked lists
- pattern matching (and regular expression)
- definition of new mini-languages: DSL
- metaprogramming: computer programs writing or manipulating other programs (or themselves) as data

To go further

- Practical Common Lisp
- Why I love Common Lisp and hate Java
- Lisp tutorial

Bonus: How to choose a programming language?

1. Figure out what works for the team/company: Media: Ruby, PHP, JS, Java; Enterprise: Java, C#; Research: Scala (simplified Java), C++, Erlang, Java and Python
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Bonus: How to choose a programming language?

1. Figure out what works for the team/company:
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Choose a Programming Language

Bonus: How to choose a programming language?

1. Figure out what works for the team/company:
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Choose a Programming Language

Bonus: How to choose a programming language?

1. Figure out what works for the team/company:
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Choose a Programming Language

Bonus: How to choose a programming language?

1. Figure out what works for the team/company:
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Choose a Programming Language

Bonus: How to choose a programming language?

1. Figure out what works for the team/company:
2. Find out what works in context
3. Consider ease of learning and use for computer programming languages
4. Evaluate the availability of tools (or libraries) for each of your potential computer programming languages
5. Look at cross-platform ability
6. Determine the ease of server-side and client-side scripting

Choose a Programming Language