Python for Lisp Programmers

This is a brief introduction to Python for Lisp programmers. (Although it wasn't my intent, Python programers have told me this page has helped them learn Lisp.) Basically, Python can be seen as a dialect of Lisp with "traditional" syntax (what Lisp people call "infix" or "m-lisp" syntax). One message on comp.lang.python said "I never understood why LISP was a good idea until I started playing with python." Python supports all of Lisp's essential features except macros, and you don't miss macros all that much because it does have eval, and operator overloading, and regular expression parsing, so some--but not all--of the use cases for macros are covered.

I looked into Python because I was considering translating the Lisp code for the Russell & Norvig AI textbook into Java. Some instructors and students wanted Java because

1. That's the language they're most familiar with from other courses.
2. They want to have graphical applications.
3. A minority want applets in browsers.
4. Some just couldn't get used to Lisp syntax in the limited amount of class time they had to devote to it.

However, our first attempt at writing Java versions was largely unsuccesful. Java was too verbose, and the differences between the pseudocode in the book and the Java code was too large. I looked around for a language that was closer to the pseudocode in the book, and discovered Python was the closest. Furthermore, with and Jython, I could target the Java JVM.

My conclusion

Python is an excellent language for my intended use. It is easy to use (interactive with no compile-link-load-run cycle), which is important for my pedagogical purposes. While Python doesn't satisfy the prerequisite of being spelled J-A-V-A, Jython is close. Python seems to be easier to read than Lisp for someone with no experience in either language. The Python code I developed looks much more like the (independently developed) pseudo-codein the book than does the Lisp code. This is important, because some students were complaining that they had a hard time seeing how the pseudo-code in the book mapped into the online Lisp code (even though it seemed obvious to Lisp programmers).

The two main drawbacks of Python from my point of view are (1) there is very little compile-time error analysis and type declaration, even less than Lisp, and (2) execution time is much slower than Lisp, often by a factor of 10 (sometimes by 100 and sometimes by 1). Qualitatively, Python feels about the same speed as interpreted Lisp, but very noticably slower than compiled Lisp. For this reason I wouldn't recommend Python for applications that are (or are likely to become over time) compute intensive (unless you are willing to move the speed bottlenecks into C). But my purpose is oriented towards pedagogy, not production, so this is less of an issue.

Introducing Python

Python can be seen as either a practical (better libraries) version of Scheme, or as a cleaned-up (no $@&% characters) version of Perl. While Perl's philosophy is TIMTOWTDI (there's more than one way to do it), Python tries to provide a minimal subset that people will tend to use in the same way (maybe TOOWTDI for there's only one way to do it, but of course there's always more than one way if you try hard). One of Python's controversial features, using indentation level rather than begin/end or braces, was driven by this philosophy: since there are no braces, there are no style wars over where to put the braces. Interestingly, Lisp has exactly the same philosphy on this point: everyone uses emacs to indent their code, so they don't argue over the indentation. Take a Lisp program, indent it properly, and delete the opening parens at the start of lines and their matching close parens, and you end up with something that looks rather like a Python program.

Python has the philosophy of making sensible compromises that make the easy things very easy, and don't preclude too many hard things. In my opinion it does a very good job. The easy things are easy, the harder things are progressively harder, and you tend not to notice the inconsistencies. Lisp has the philosophy of making fewer compromises: of providing a very powerful and totally consistent core. This can make Lisp harder to learn because you operate at a higher level of abstraction right from the start and because you need to understand what you're doing, rather than just relying on what feels or looks nice. But it also means that in Lisp it is easier to add levels of abstraction and complexity; Lisp is optimized to make the very hard things not too hard, while Python is optimized to make medium hard things easier.

Here I've taken a blurb from Python.org and created two vesions of it: one for*Python in blue italics* and one for **Lisp in green bold**. The bulk of the blurb, common to both languages, is in black.

> *Python*/**Lisp** is an interpreted **and compiled**, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. *Python*/**Lisp**'s simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. *Python*/**Lisp** supports*modules and* packages, which encourages program modularity and code reuse. The *Python*/**Lisp** interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. Often, programmers fall in love with *Python*/**Lisp** because of the increased productivity it provides. Since there is no **separate**compilation step, the edit-test-debug cycle is incredibly fast. Debugging *Python*/**Lisp** programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in *Python*/**Lisp** itself, testifying to*Python*/**Lisp**'s introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective.

To which I can only add:

> Although some people have initial resistance to the *indentation as block structure*/**parentheses**, most come to *like*/**deeply appreciate** them.

To learn more about Python, if you are an experienced programmer, I recommend going to the download page at Python.org and getting the documentation package, and paying particular attention to the Python Reference Manual and the Python Library Reference. There are all sorts of tutorials and published books, but these references are what you really need.

The following table serves as a Lisp/Python translation guide. Entries in redmark places where one language is noticibly worse, in my opinion. Entries in**bold** mark places where the languages are noticibly different, but neither approach is clearly better. Entries in regular font mean the languages are similar; the syntax might be slightly different, but the concepts are the same or very close. The table is followed by a list of gotchas and some sample programs in Python.

Key Features	**Lisp Features**	**Python Features**
Everything is an object	Yes	Yes
Objects have type, variables don't	Yes	Yes
Support heterogeneous lists	Yes (**linked list** and array/vector)	Yes (array)
Multi-paradigm language	Yes: Functional, Imperative, OO,**Generic**	Yes: Functional, Imperative, OO
Storage management	Automatic garbage collection	Automatic garbage collection
Packages/Modules	Harder to use	Easy to use
Introspection of objects, classes	Strong	Strong
Macros for metaprogramming	Powerful macros	No macros
Interactive Read-eval-print loop	> (string-append "hello" " " "world") "hello world"	>>> ' '.join(['hello', 'world']) 'hello world'
Concise expressive language	(defun transpose (m) (apply #'mapcar #'list m)) > (transpose '((1 2 3) (4 5 6))) ((1 4) (2 5) (3 6))	def transpose (m): return zip(*m) >>> transpose([[1,2,3], [4,5,6]]) [(1, 4), (2, 5), (3, 6)]
Cross-platform portability	Windows, Mac, Unix, Gnu/Linux	Windows, Mac, Unix, Gnu/Linux
Number of implementations	**Many**	**One** main, plus branches (e.g. Jython, Stackless)
Development Model	Proprietary and open source	Open source
Efficiency	About 1 to 2 times slower than C++	About 2 to 100 times slower than C++
GUI, Web, etc. librariees	Not standard	GUI, Web libraries standard
Methods Method dispatch	Dynamic, **(meth obj arg)** syntax runtime-type, **multi-methods**	Dynamic, **obj.meth(arg)** syntax runtime-type, **single class-based**
Data Types	**Lisp Data Types**	**Python Data Types**
Integer	42	42
Bignum	100000000000000000	100000000000000000
Float	12.34	12.34
Complex	#C(1, 2)	1 + 2J
String	"hello"	"hello" *or* 'hello' ## **immutable**
Symbol	hello	'hello'
Hashtable/Dictionary	(make-hash-table)	{}
Function	(lambda (x) (+ x x))	lambda x: x + x
Class	(defclass stack ...)	class Stack: ...
Instance	(make 'stack)	Stack()
Stream	(open "file")	open("file")
Boolean	t, nil	True, False
Empty Sequence	(), #() *linked list, array*	(), [] *tuple, array*
Missing Value	nil	None
Lisp List (linked)	(1 2.0 "three")	(1, (2.0, ("three", None)))
Python List (adjustable array)	(make-array 3 :adjustable t :initial-contents '(1 2 3))	[1, 2.0, "three"]
Others	Many (in core language)	Many (in libraries)
Control Structures	**Lisp Control Structures**	**Python Control Structures**
Statements and expressions	**Everything is an expression**	**Distinguish statements from expressions**
False values	**nil** is only false value	**False, None, 0, '', [ ], {}** are all false
Function call	(func x y z)	func(x,y,z)
Conditional test	(if x y z)	if x: y else: z
Conditional expression	(if x y z)	y if x else z
While loop	(loop while (test) do (f))	while test(): f()
Other loops	(dotimes (i n) (f i)) (loop for x in s do (f x))	for i in range(n): f(i) for x in s: f(x) ## *works on any sequence*

	(loop for (name addr salary) in db do ...)	for (name, addr, salary) in db: ...
Assignment	(setq x y) (psetq x 1 y 2) (rotatef x y) (setf (slot x) y) **(values 1 2 3)** *on stack* (multiple-value-setq (x y) (values 1 2))	x = y x, y = 1, 2 x, y = y, x x.slot = y **(1, 2, 3)** *uses memory in heap* x, y = 1, 2
Exceptions	(assert (/= denom 0)) (unwind-protect (attempt) (recovery)) (catch 'ball ... (throw 'ball))	assert denom != 0, "denom != 0" try: attempt() finally: recovery() try: ...; raise 'ball' except 'ball': ...
Other control structures	case, etypecase, cond, with-open-file,*etc.*	Extensible **with** statement **No other control structures**
Lexical Structure	**Lisp Lexical Structure**	**Python Lexical Structure**
Comments	;; semicolon to end of line	## hash mark to end of line

Delimiters	**Parentheses to delimit expressions:**
	(defun fact (n)
	  (if (<= n 1) 1
	      (* n (fact (- n 1)))))
		Indentation to delimit statements:
		def fact (n):
		    if n <= 1: return 1
		    else: return n * fact(n — 1)

Higher-Order Functions	**Lisp Higher-Order Functions**	**Python Higher-Order Functions**
Function application evaluate an expression execute a statement load a file	(apply fn args) (eval '(+ 2 2)) => 4 (eval '(dolist (x list) (f x))) (load "file.lisp") *or* (require 'file)	apply(fn, args) *or* fn(*args) eval("2+2") => 4 exec("for x in list: f(x)") execfile("file.py") *or* import file
Sequence functions	(mapcar length '("one" (2 3))) => (3 2) (reduce #'+ numbers) (every #'oddp '(1 3 5)) => T (some #'oddp '(1 2 3)) => **1** (remove-if-not #'evenp numbers) (reduce #'min numbers)	map(len, ["one", [2, 3]]) => [3, 2] *or* [len(x) for x in ["one", [2, 3]]] reduce(operator.add, numbers) all(x%2 for x in [1,3,5]) => True any(x%2 for x in [1,2,3]) => **True** filter(lambda x: x%2 == 0, numbers) *or* [x for x in numbers if x%2 == 0] min(numbers)
Other higher-order functions	count-if, *etc.* :test, :key, etc keywords	No other higher-order functions built-in No keywords on map/reduce/filter
Close over read-only var Close over writable var	(lambda (x) (f x y)) (lambda (x) (incf y x))	lambda x: f(x, y) Can't be done; use objects
Parameter Lists	**Lisp Parameter Lists**	**Python Parameter Lists**

Optional arg
Variable-length arg
Unspecified keyword args
Calling convention
	(defun f (&optional (arg val) ...)
	(defun f (&rest arg) ...)
	(defun f (&allow-other-keys &rest arg) ...)
	Call with keywords only when declared:
	(defun f (&key x y) ...)
	(f :y 1 :x 2)
		def f (arg=val): ...
		def f (*arg): ...
		def f (**arg): ...
		Call any function with keywords:
		def f (x,y): ...
		f(y=1, x=2)

Efficiency	**Lisp Efficiency Issues**	**Python Efficiency Issues**
Compilation Function reference resolution Declarations	Compiles to native code Most function/method lookups are fast Declarations can be made for efficiency	Compiles to bytecode only Most function/method lookups are slow No declarations
Features	**Lisp Features and Functions**	**Python Features and Functions**
Quotation	**Quote whole list structure:** 'hello '(this is a test) '(hello world (+ 2 2))	**Quote individual strings** or `.split()`: 'hello' 'this is a test'.split() ['hello', 'world', [2, "+", 2]]

Introspectible doc strings	(defun f (x)
	  "compute f value"
	  ...)
	> (documentation 'f 'function)
	"compute f value"
		def f(x):
		  "compute f value"
		  ...
		>>> f.__doc__
		"compute f value"

List access	**Via functions:** (first list) (setf (elt list n) val) (first (last list)) (subseq list start end) (subseq list start)	**Via syntax:** list[0] list[n] = val list[-1] list[start:end] list[start:]

Hashtable access	**Via functions:** (setq h (make-hash-table)) (setf (gethash "one" h) 1.0) (gethash "one" h) (let ((h (make-hash-table))) (setf (gethash "one" h) 1) (setf (gethash "two" h) 2) h)	**Via syntax:** h = {} h["one"] = 1.0 h["one"] *or* h.get("one") h = {"one": 1, "two": 2}
Operations on lists	(cons x y) (car x) (cdr x) (equal x y) (eq x y) nil (length seq) (vector 1 2 3)	[x] + y *but O(n); also* y.append(x) x[0] x[1:] *but O(n)* x == y x is y () *or* [ ] len(seq) (1, 2, 3)

Operations on arrays	(make-array 10 :initial-element 42)
	(aref x i)
	(incf (aref x i))
	(setf (aref x i) 0)
	(length x)
	#(10 20 30) *if size unchanging*
		10 * [42]
		x[i]
		x[i] += 1
		x[i] = 0
		len(x)
		[10, 20, 30]

An important point for many people is the speed of Python and Lisp versus other languages. Its hard to get benchmark data that is relevent to *your* set of applications, but this may be useful:

Test	Lisp	Java	Python	Perl	C++
hash access	1.06	3.23	4.01	1.85	1.00
exception handling	0.01	0.90	1.54	1.73	1.00
sum numbers from file	7.54	2.63	8.34	2.49	1.00
reverse lines	1.61	1.22	1.38	1.25	1.00
matrix multiplication	3.30	8.90	278.00	226.00	1.00
heapsort	1.67	7.00	84.42	75.67	1.00
array access	1.75	6.83	141.08	127.25	1.00
list processing	0.93	20.47	20.33	11.27	1.00
object instantiation	1.32	2.39	49.11	89.21	1.00
word count	0.73	4.61	2.57	1.64	1.00
Median	1.67	4.61	20.33	11.27	1.00
25% to 75%	0.93 to 1.67	2.63 to 7.00	2.57 to 84.42	1.73 to 89.21	1.00 to 1.00
Range	0.01 to 7.54	0.90 to 20.47	1.38 to 278	1.25 to 226	1.00 to 1.00

Legend
> 100 x C++
50-100 x C++
10-50 x C++
5-10 x C++
2-5 x C++
1-2 x C++
< 1 x C++

Relative speeds of 5 languages on 10 benchmarks from The Great Computer Language Shootout.

Speeds are normalized so the g++ compiler for C++ is 1.00, so 2.00 means twice as slow; 0.01 means 100 times faster. For Lisp, the CMUCL compiler was used. Background colors are coded according to legend on right. The last three lines give the mean score, 25% to 75% quartile scores (throwing out the bottom two and top two scores for each language), and overall range. Comparing Lisp and Python and throwing out the top and bottom two, we find Python is 3 to 85 times slower than Lisp -- about the same as Perl, but much slower than Java or Lisp. Lisp is about twice as fast as Java.

Gotchas for Lisp Programmers in Python

Here I list conceptual problems for me as a Lisp programmer coming to Python:

1. **Lists are not Conses.** Python lists are actually like adjustable arrays in Lisp or Vectors in Java. That means that list access is *O(1)*, but that the equivalent of both cons and cdr generate *O(n)* new storage. You really want to use `map` or `for e in x:` rather than car/cdr recursion. Note that there are multiple empty lists, not just one. This fixes a common bug in Lisp, where users do `(nconc old new)` and expect `old` to be modified, but it is not modified when `old` is `nil`. In Python, `old.extend(new)`works even when `old` is `[ ]`. But it does mean that you have to test against `[]` with `==`, not `is`, and it means that if you set a default argument equal to `[]` you better not modify the value.
2. **Python is less functional.** Partially because lists are not conses, Python uses more methods that mutate list structure than Lisp, and to emphasize that they mutate, they tend to return `None`. This is true for methods like`list.sort`, `list.reverse`, and `list.remove`. However, more recent versions of Python have snuck functional versions back in, as functions rather than methods: we now have `sorted` and `reversed` (but not`removed`).
3. **Python classes are more functional.** In Lisp (CLOS), when you redefine a class C, the object that represents C gets modified. Existing instances and subclasses that refer to C are thus redirected to the new class. This can sometimes cause problems, but in interactive debugging this is usually what you want. In Python, when you redefine a class you get a new class object, but the old instances and subclasses still refer to the old class. This means that most of the time you have to reload your subclasses and rebuild your data structures every time you redefine a class. If you forget, you can get confused.
4. **Python is more dynamic, does less error-checking**. In Python you won't get any warnings for undefined functions or fields, or wrong number of arguments passed to a function, or most anything else at load time; you have to wait until run time. The commercial Lisp

implementations will flag many of these as warnings; simpler implementations like clisp do not. The one place where Python is demonstrably more dangerous is when you do `self.feild = 0` when you meant to type `self.field = 0`; the former will dynamically create a new field. The equivalent in Lisp, `(setf (feild self) 0)` will give you an error. On the other hand, accessing an undefined field will give you an error in both languages.

5. **Don't forget self.** This is more for Java programmers than for Lisp programmers: within a method, make sure you do `self.field`, not`field`. There is no implicit scope. Most of the time this gives you a run-time error. It is annoying, but I suppose one learns not to do it after a while.
6. **Don't forget return.** Writing `def twice(x): x+x` is tempting and doesn't signal a warning or exception, but you probably meant to have a`return` in there. This is particularly irksome because in a `lambda` you are prohibited from writing `return`, but the semantics is to do the return.
7. **Watch out for singleton tuples.** A tuple is just an immutable list, and is formed with parens rather than square braces. `()` is the empty tuple, and`(1, 2)` is a two-element tuple, but `(1)` is just 1. Use `(1,)` instead. Yuck. Damian Morton pointed out to me that it makes sense if you understand that tuples are printed with parens, but that they are formed by commas; the parens are just there to disambiguate the grouping. Under this interpretation, `1, 2` is a two element tuple, and `1,` is a one-element tuple, and the parens are sometimes necessary, depending on where the tuple appears. For example, `2, + 2,` is a legal expression, but it would probably be clearer to use `(2,) + (2,)` or `(2, 2)`.
8. **Watch out for certain exceptions.** Be careful: `dict[key]` raises`KeyError` when `key` is missing; Lisp hashtable users expect `nil`. You need to catch the exception or test with `key in dict`.
9. **Python is a Lisp-1.** By this I mean that Python has one namespace for functions and variables, like Scheme, not two like Common Lisp. For example:

```
def f(list, len): return list((len, len(list)))      ## bad Python
(define (f list length) (list length (length list))) ;; bad Scheme
(defun f (list length) (list length (length list)))  ;; legal Common Lisp
```

   This also holds for fields and methods: you can't provide an abstraction level over a field with a method of the same name:

```
class C:
    def f(self): return self.f  ## bad Python
    ...
```

10. **Python strings are not quite like Lisp symbols.** Python does symbol lookup by interning strings in the hash tables that exist in modules and in classes. That is, when you write `obj.slot` Python looks for the string`"slot"` in the hash table for the class of `obj`, at run time. Python also interns some strings in user code, for example when you say `x = "str"`. But it does not intern strings that don't look like variables, as in `x = "a str"` (thanks to Brian Spilsbury for pointing this out).
11. **Python does not have macros.** Python does have access to the abstract syntax tree of programs, but this is not for the faint of heart. On the plus side, the modules are easy to understand, and with five minutes and five lines of code I was able to get this:

```
>>> parse("2 + 2")
['eval_input', ['testlist', ['test', ['and_test', ['not_test', ['comparison',
 ['expr', ['xor_expr', ['and_expr', ['shift_expr', ['arith_expr', ['term',
  ['factor', ['power', ['atom', [2, '2']]]]], [14, '+'], ['term', ['factor',
   ['power', ['atom', [2, '2']]]]]]]]]]]]]]], [4, ''], [0, '']]
```

   This was rather a disapointment to me. The Lisp parse of the equivalent expression is `(+ 2 2)`. It seems that only a real expert would want to manipulate Python parse trees, whereas Lisp parse trees are simple for anyone to use. It is still possible to create something similar to macros in Python by concatenating strings, but it is not integrated with the rest of the language, and so in practice is not done. In Lisp, there are two main purposes for macros: new control structures, and custom problem-specific languages. The former is just not done in Python. The later can be done for *data* with a problem-specific format in Python: below I define a context-free grammar in Python using a combination of the builtin syntax for dictionaries and a preprocessing step that parses strings into data structures. The combination is almost as nice as Lisp macros. But more complex tasks, such as writing a compiler for a logic programming language, are easy in Lisp but hard in Python.

Comparing Lisp and Python Programs

I took the first example program from Paradigms of Artificial Intelligence Programming, a simple random sentence generator and translated it into Python. Conclusions: conciseness is similar; Python gains because`grammar[phrase]` is simpler than `(rule-rhs (assoc phrase *grammar*))`, but Lisp gains because `'(NP VP)` beats `['NP', 'VP']`. The Python program is probably less efficient, but that's not the point. Both languages seem very well suited for programs like this. *Make your browser window wide to see this properly.*

Lisp Program simple.lisp

```
(defparameter *grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
         (mappend #'generate phrase))
        ((rewrites phrase)
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))))

(defun generate-tree (phrase)
  "Generate a random sentence or phrase,
  with a complete parse tree."
```

Python Program simple.py

```
from random import choice

grammar = dict(
        S = [['NP','VP']],
        NP = [['Art', 'N']],
        VP = [['V', 'NP']],
        Art = ['the', 'a'],
        N = ['man', 'ball', 'woman', 'table'],
        V = ['hit', 'took', 'saw', 'liked']
        )

def generate(phrase):
    "Generate a random sentence or phrase"
    if isinstance(phrase, list):
        return mappend(generate, phrase)
    elif phrase in grammar:
        return generate(choice(grammar[phrase]))
    else: return [phrase]

def generate_tree(phrase):
```

```lisp
  (cond ((listp phrase)
         (mapcar #'generate-tree phrase))
        ((rewrites phrase)
         (cons phrase
               (generate-tree (random-elt (rewrites phrase)))))
        (t (list phrase))))

(defun mappend (fn list)
  "Append the results of calling fn on each element of list.
  Like mapcon, but uses append instead of nconc."
  (apply #'append (mapcar fn list)))

(defun rule-rhs (rule)
  "The right hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))
```

```python
    """Generate a random sentence or phrase,
     with a complete parse tree."""
    if isinstance(phrase, list):
        return map(generate_tree, phrase)
    elif phrase in grammar:
        return [phrase] + generate_tree(choice(grammar[phrase]))
    else: return [phrase]

def mappend(fn, list):
    "Append the results of calling fn on each element of list."
    return reduce(lambda x,y: x+y, map(fn, list))
```

Running the Lisp Program

```
> (generate 'S)
(the man saw the table)
```

Running the Python Program

```
>>> generate('S')
['the', 'man', 'saw', 'the', 'table']

>>> ' '.join(generate('S'))
'the man saw the table'
```

I was concerned that the grammar is uglier in Python than in Lisp, so I thought about writing a Parser in Python (it turns out there are some already written and freely available) and about overloading the builtin operators. This second approach is feasible for some applications, such as my Expr class for representing and manipulating logical expressions. But for this application, a trivial ad-hoc parser for grammar rules will do: a grammar rule is a list of alternatives, separated by '|', where each alternative is a list of words, separated by ' '. That, plus rewriting the grammar program in idiomatic Python rather than a transliteration from Lisp, leads to the following program:

Python Program simple.py (idiomatic version)

```python
"""Generate random sentences from a grammar.  The grammar
consists of entries that can be written as S = 'NP VP | S and S',
which gets translated to {'S': [['NP', 'VP'], ['S', 'and', 'S']]}, and
means that one of the top-level lists will be chosen at random, and
then each element of the second-level list will be rewritten; if a symbol is
not in the grammar it rewrites as itself.   The functions generate and
generate_tree generate a string and tree representation, respectively, of
a random sentence."""

import random

def Grammar(**grammar):
  "Create a dictionary mapping symbols to alternatives."
  for (cat, rhs) in grammar.items():
    grammar[cat] = [alt.split() for alt in rhs.split('|')]
  return grammar

grammar = Grammar(
  S  = 'NP VP',
  NP = 'Art N',
  VP = 'V NP',
  Art= 'the | a',
  N  = 'man | ball | woman | table',
  V  = 'hit | took | saw | liked'
  )

def generate(symbol='S'):
  "Replace symbol with a random entry in grammar (recursively); join into a string."
  if symbol not in grammar:
    return symbol
  else:
    return ' '.join(map(generate, random.choice(grammar[symbol])))

def generate_tree(symbol='S'):
  "Replace symbol with a random entry in grammar (recursively); return a tree."
  if symbol not in grammar:
    return symbol
  else:
    return {symbol: map(generate_tree, random.choice(grammar[symbol]))}
```

This page also available in Japanese translation by Yusuke Shinyama.

Peter Norvig