Python Supernova

Robert Keller

December 2012
How to Learn a New Language?

learnLanguage(L) :-
    findExample(L, E),
    imitateExample(E),
    fail.
learnLanguage(L) :-
    acquireManual(L, M),
    readManual(M),
    fail.
learnLanguage(L) :-
    findYouTubeVideo(L, V),
    viewVideo(V),
    fail.
learnLanguage(L) :-
    askForum(L, P),
    viewAnswer(P),
    fail.
learnLanguage(L) :-
    tryToTeach(L).
How to Jump Start Learning a Language

• **Imitate** existing examples.

• Try to align them with **concepts you already know** from other languages.

• There are 1000’s of languages, but probably fewer than 100 concepts.

• Try to “see through” the syntax to the core.

• Determine what data types are available.

• Try to discern what the underlying data and control models are.
# How to Jump Start Learning a Language

<table>
<thead>
<tr>
<th>Concept</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open lists</td>
<td>Racket, Prolog</td>
</tr>
<tr>
<td>Function</td>
<td>Racket</td>
</tr>
<tr>
<td>Anonymous function</td>
<td>Racket</td>
</tr>
<tr>
<td>Object</td>
<td>Java</td>
</tr>
<tr>
<td>Method</td>
<td>Java</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>Prolog</td>
</tr>
<tr>
<td>Predicate logic</td>
<td>Prolog, SQL</td>
</tr>
<tr>
<td>2D matrices</td>
<td>Matlab</td>
</tr>
<tr>
<td>Infinite lists</td>
<td>Haskell</td>
</tr>
<tr>
<td>Iterators</td>
<td>C++, Java</td>
</tr>
<tr>
<td>List comprehensions</td>
<td>KRC, Miranda, Haskell</td>
</tr>
<tr>
<td>Type inference</td>
<td>ML</td>
</tr>
</tbody>
</table>
High vs. Low Ceremony Languages

• **High Ceremony** (classes, header files, …)
  Java, C, C++, Ada, Fortran, …

• **Low Ceremony:**
  Racket, Prolog, Matlab, Python, Ruby, Javascript, Groovy, Lua, Basic, …

• High ceremony tend to require a compiler, whereas low ceremony sometimes just use an interpreter, a REPL.

• “*Scripting*” languages tend to be low ceremony. They often include useful built-ins for interacting with the operating system.
Static vs. Dynamic Typing

- **Static** vs. **Dynamic** refers to whether a variable can refer only one type of thing (int, String, function, …) or multiple types.

- Racket and Prolog are examples of dynamic languages.

- Java is (mostly) statically typed, although using Object as a type allows dynamic as well.

- ML, Haskell, Scala, and go are statically typed.
Type Safety

• Static types allow the compiler to check for type errors in advance.

• With dynamic types, errors might not be detected until the program runs.

• But static types often require more ceremony.

• Dynamic types may be preferred for prototyping, static types for production.

• Static typing tends to go with higher ceremony.
Python

Download python here: http://www.python.org/

Alternative Implementations

This site hosts the "traditional" implementation of Python (nicknamed CPython). A number of alternative implementations are available as well, namely

- IronPython (Python running on .NET)
- Jython (Python running on the Java Virtual Machine)
- PyPy (A fast python implementation with a JIT compiler)
- Stackless Python (Branch of CPython supporting microthreads)

Download Python

The current production versions are **Python 2.7.3 and Python 3.3.0**.

Start with one of these versions for learning Python or if you want the most stability; they're both considered stable production releases.

If you don't know which version to use, start with Python 2.7; more existing third party software is compatible with Python 2 than Python 3 right now.

For the MD5 checksums and OpenPGP signatures, look at the detailed Python 2.7.3 page:
Python Discontinuity

• Python 2.7.x
• Python 3.3.x

apparently are different enough that most libraries run only on the first version.
Python Characteristics

• Python is an imperative language with some support for objects, modules, and functional programming.

• Dynamically typed

• Interpreted, REPL

• Scripting
Python Script

• Scripts run as OS commands on the command line.
• They identify the location of the Python interpreter.

Definition of pi.py:

```python
#!/Library/Frameworks/Python.framework/Versions/2.7/bin/python
from math import pi
print(pi)
```

Example use (in Unix):

```
$ ./pi.py
3.14159265359
```
Script with Command-Line Arguments

• `sys.argv` identifies the items on the command line, starting with the command itself

contents of sort.py

```python
#!/Library/Frameworks/Python.framework/Versions/2.7/bin/python
import sys
args = sys.argv
args.pop(0)
args.sort()

for x in args:
    print(x)
```

Example Use:
```
$ ./sort.py 5 4 2 1 3
1
2
3
4
5
```
Python Strings

- Use matching single or double quotes
- Evaluate for value
  >>> 'foo bar'
  'foo bar'

  >>> "foo bar"
  'foo bar'

- Ways to embed quotes in string:
  >>> "doesn't"
  "doesn't"

  >>> 'doesn"t'
  'doesn"t'

  >>> 'doesn\'t'
  'doesn\'t'

  >>> 'doesn\"t'
  'doesn\"t'

  >>> 'doesn\"t'
  'doesn\"t'
Assignment and Recall

```python
>>> z = "abc"
>>> z
'abc'

>>> x = 12371289371827398172839172837912738127381973
>>> x
12371289371827398172839172837912738127381973L

>>> y = 34.567e13
>>> y
345670000000000.0

>>> x*y
4.2763835971595765e+57
```
Python Data Types

- Integers (arbitrary precision)
- Float (double precision)
- Complex (use .real and .imag to get components)
- Strings
- Lists
- Tuples
- Dictionaries
- Objects
Lists

• Lists are mutable

• Lists are rendered comma-separated, in square brackets.
  
  >>> ['This', 'is', 'a', 'list.]
  
  ['This', 'is', 'a', 'list.]

• Indexing starts at 0, and also uses square brackets.
  
  >>> ['This', 'is', 'a', 'list.'][2]
  
  'a'
Nested Lists

• Lists can be nested, as in Racket and Prolog:

```python
>>> ['This', ['is', ['a', ['nested', 'list.']]]]
['This', ['is', ['a', ['nested', 'list.']]]]
```
Lists are Mutable

Indexing starts at 0

```python
>>> L = [10, 20, 30, 40]

>>> L[0]
10

>>> L[1]
20

>>> L[3] = 42

>>> L
[10, 20, 30, 42]
```
>>> L
[10, 20, 30, 42]

>>> L[-1]
42

>>> L[-2]
30

>>> L[-3]
20
List Operations

- **len(list)**
  Returns the length of the list.

- **list.append(x)**
  Add an item to the end of the list. This is like `cons` in Racket, except on the tail of the list, instead of the head.

- **list.extend(L)**
  Extend the list by appending all the items in the given list. This is like `append` in OpenList.

- **list.insert(i, x)**
  Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

- **list.remove(x)**
  Remove the first item from the list whose value is `x`. It is an error if there is no such item.
More List Operations

• **list.pop(i)**
  Remove the item at the given position in the list, and return it.
  If no index is specified, `a.pop()` removes and returns the last item in the list.

• **list.index(x)**
  Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

• **list.count(x)**
  Return the number of times `x` appears in the list.

• **list.sort()**
  Sort the items of the list, **in place**.

• **list.reverse()**
  Reverse the elements of the list, **in place**.
Tuples

- Tuples are immutable
- Tuples are rendered as comma-separated, with regular parentheses.

```python
>>> type(['a', 'b', 'c'])
<type 'list'>

>>> type(('a', 'b', 'c'))
<type 'tuple'>
```
Tuples and Lists can be Interconverted

```python
>>> [x,y,z]=(1,2,3)
>>> [x,y,z]
[1, 2, 3]

>>> (a, b, c) = [x, y, z]
>>> (a, b, c)
(1, 2, 3)
```
Dictionaries

• Curly braces \{ \} represent the **dictionary** data type.

• They are not used for code organization, as in Java and C.

• A **dictionary** is a hashtable, and can serve a role similar to an association list.

• Square brackets \[ \] are used to access elements of a dictionary
Dictionary Example

```python
>>> my_dictionary = {'a':10, 'b':20, 'c':30}

>>> my_dictionary
{'a': 10, 'c': 30, 'b': 20}

>>> my_dictionary['b']
20
```
Dictionaries are Mutable

```python
>>> my_dictionary
{'a': 10, 'c': 30, 'b': 20}

>>> my_dictionary['b'] = 42

>>> my_dictionary
{'a': 10, 'c': 30, 'b': 42}
```
Global Variables

- Assignments shown bind variables in the global namespace

- You can find out what variables are bound by the built-in `globals()`, which returns a dictionary

```python
>>> globals()
{'a': 'foo bar', 'hello': <module 'hello' from 'hello.py'>, 'x': 12371289371827398172839172837912738127381973L, 'y': 345670000000000.0, 'z': 'abc', ...}
```
Indentation

- Indentation is mandatory in Python control structures.
- It replaces braces in Java.
- In the REPL, indent 4 spaces after …, which tell you that the command is continuing.
- Colons after the if part and else part are required.

```python
>>> if x > 4/3:
    print('yes')
else:
    print('no')
... yes
```
elif = else + if

```python
>>> if x < 4/3:
...     print('yes')
... elif x == 0:
...     print('maybe')
... else:
...     print('no')

no
```
```python
>>> i = 1
>>> while i < 10:
...     print(i)
...     i = i + 1
...
1
2
3
4
5
6
7
8
9
```
for

• *for* iterates over a list

```python
>>> for color in ['red', 'green', 'blue']:
...     print(color)
...     print(color)
...
red
green
blue
```
range

• range makes a list, e.g. to be used with for

```python
>>> range(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```python
>>> range(2, 11, 2)
[2, 4, 6, 8, 10]
```

```python
>>> range(11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
for in range

```python
>>> for x in range(0, 10):
...     print(x)
...
...
0
1
2
3
4
5
6
7
8
9
>>> x
9
```
List Comprehensions

- A clever idea that came from functional languages (KRC & Miranda by David Turner, University of Kent)

- Recall set notation from math:
  \{f(x) \mid \ldots \text{some property of } x \ldots \}\)

- Analogously, a list can be constructed
  \[f(x) \ldots \text{some iterator of } x \ldots \]

- Advantage: More declarative

- Example:
  \[x^2 \text{ for } x \text{ in range}(1, 11)\]
  is \[1, 4, 9, 16, 25, 36, 49, 64, 81, 100\]
List Comprehension Examples

>>> [x*x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> def mymap(f, L):
...     return [f(x) for x in L]

>>> mymap(lambda x: x*x, range(1, 11))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> [(x, y) for x in [1, 2, 3] for y in ['a', 'b', 'c']]  
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]

>>> def cartesian_product(L, M):
...     [(x, y) for x in L for y in M]
...     return [(x, y) for x in L for y in M]

>>> cartesian_product(range(1, 4), ['a', 'b', 'c'])
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
Using *if* in List Comprehension

- See example under Class Definition
>>> def fac(n):
...     if n < 2:
...         return 1
...     return n*fac(n-1)
...

>>> fac
<function fac at 0x100499c80>

>>> fac(5)
120
Function Definitions and Docstrings

Finally, a use for triple double quotes

Note that indentation is required, not optional.

```python
def add(x, y):
    """Return the sum of x and y.""
    return x + y
```

The docstring is the first string literal to appear in an object's definition. An object's docstring shows up when you use the built-in "help" function:

```
>>> help(add)
Help on function add in module __main__:

add(x, y)
    Return the sum of x and y.
```

You can look up any object's docstring via the `__doc__` attribute:

```python
>>> print add.__doc__
Return the sum of x and y.
```
Anonymous Functions

```python
>>> square = lambda x: x*x

>>> square
<function <lambda> at 0x100499d70>

>>> square(5)
25

>>> z = 100
>>> scale = lambda x: x*z
>>> scale
<function <lambda> at 0x100499cf8>

>>> scale(5)
500

>>> z = 10
>>> scale(5)
50
```
>>> def foo(x):
...     z = 99
...     return z * x
...

>>> foo(10)
990

>>> z = 90

>>> foo(10)
990
Higher Order Functions

```python
>>> def compose(f, g):
...    return lambda x: f(g(x))
...

>>> compose(scale, square)(5)
250
```
map/reduce (like foldl)

```python
>>> map
<built-in function map>

```map

```python
>>> map(square, range(1, 11))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
15

Must use lambda, not plain +.
```
Modules: Code Organization

- Modules are identified by their file name.

```
$ cat hello.py
def say_hello():
    print 'hello'
    return

$ python
>>> import hello
>>> hello.say_hello()
hello
```
Locating Modules

When you import a module, the Python interpreter searches for the module in the following sequences:

- The current directory.
- If the module isn’t found, Python then searches each directory in the shell variable PYTHONPATH.
- If all else fails, Python checks the default path. On UNIX, this default path is normally /usr/local/lib/python/.

The module search path is stored in the system module sys as the `sys.path` variable. The `sys.path` variable contains the current directory, PYTHONPATH, and the installation-dependent default.

The **PYTHONPATH** Variable:

The PYTHONPATH is an environment variable, consisting of a list of directories. The syntax of PYTHONPATH is the same as that of the shell variable PATH.

Here is a typical PYTHONPATH from a Windows system:

```
set PYTHONPATH=c:\python20\lib;
```

And here is a typical PYTHONPATH from a UNIX system:

```
set PYTHONPATH=/usr/local/lib/python
```
# Defining Classes

```python
# Student

class Student:

    instances = []  # class (static) variable

    def __init__(self, name, graduation_year):
        # constructor/initializer
        """ create an instance of Student ""
        self.name = name  # instance variable
        self.graduation_year = graduation_year  # instance variable
        self.credits = 0  # instance variable
        Student.instances.append(self)  # using class variable

    def display(self):
        # method
        """ display a Student ""
        print "Name :", self.name,
        ", Graduation_Year:, self.graduation_year,
        ", Credits:, self.credits

    def addCredit(self, units):
        # method
        """ add units to this Student's credits ""
        self.credits += units

    def getCredits(self):
        # method
        """ return the credits of this Student ""
        return self.credits

    def getCount(self):
        # method using class variable
        """ get the total number of Students ""
        return len(Student.instances)

    def getCohort(self):
        # method using class variable
        """ get the cohort of this Student as a list ""
        return [student for student in Student.instances
                  if student.graduation_year == self.graduation_year]
```

---

## Defining Classes

```python
# Student

class Student:

    instances = []  # class (static) variable

    def __init__(self, name, graduation_year):
        # constructor/initializer
        """ create an instance of Student ""
        self.name = name  # instance variable
        self.graduation_year = graduation_year  # instance variable
        self.credits = 0  # instance variable
        Student.instances.append(self)  # using class variable

    def display(self):
        # method
        """ display a Student ""
        print "Name :", self.name,
        ", Graduation_Year:, self.graduation_year,
        "", "Credits:", self.credits

    def addCredit(self, units):
        # method
        """ add units to this Student's credits ""
        self.credits += units

    def getCredits(self):
        # method
        """ return the credits of this Student ""
        return self.credits

    def getCount(self):
        # method using class variable
        """ get the total number of Students ""
        return len(Student.instances)

    def getCohort(self):
        # method using class variable
        """ get the cohort of this Student as a list ""
        return [student for student in Student.instances
                  if student.graduation_year == self.graduation_year]
```
Creating and Using Objects

```python
# Create 4 students
alice = Student("Alice", 2015)
bob   = Student("Bob", 2015)
carol = Student("Carol", 2016)
dave  = Student("Dave", 2016)

alice.display()

print "Total students:", alice.getCount()

print "Bob's cohort:",
for student in bob.getCohort():
    student.display()

print "Dave's cohort:",
for student in dave.getCohort():
    student.display()
```
Libraries

- Pygame - Game Development
- SciPy - Scientific computing
- NumPy - Numeric computing
- Pybrain - Machine Learning
- Django - Web Development
Mashups

- jython = Python interface to Java
- Cython = C extension to Python
- PyScheme = Scheme (Racket’s parent) in Python
- PySWIP = Python + SWI Prolog
- PyPHP = Python + PHP
- PySqlite = Python + SQL
- Pyrex = glassware
Welcome to PyScheme! Type: (QUIT) to quit.

[PyScheme] >>> (define (factorial x)
[......1] >>> (if (= x 0)
[......2] >>> 1
[......2] >>> (* x (factorial (- x 1)))
OK
[PyScheme] >>> (factorial 5)
120
[PyScheme] >>> (define (map f I)
[......1] >>> (cond ((null? I) '())
[......2] >>> (else (cons (f (car l))
[......4] >>> (map f (cdr l))))))
OK
[PyScheme] >>> (map factorial '(1 2 3 4 5 6))
[1, 2, 6, 24, 120, 720]
[PyScheme] >>> (define foo 'unquote)
OK
[PyScheme] >>> '(hey look I support ,foo)
['HEY', 'LOOK', 'I', 'SUPPORT', 'UNQUOTE']

;; PyScheme 1.0 stuff:
[PyScheme] >>> ((lambda (x) (list x (list (quote quote) x)))
[......1] >>> (quote (lambda (x) (list x (list (quote quote) x))))
((lambda (x) (list x (list (quote quote) x))) (quote (lambda (x) (list x (list (quote quote) x)))))
[PyScheme] >>> (append '(1 2 3) (list 4 5 '(6)) 7)
(1 2 3 4 5 (6) . 7)
[PyScheme] >>> (quit)
bye