

Python as the Pascal of 2000's

Luby Liao
Department of Mathematics/Computer Science
University of San Diego
San Diego, CA 92110
`liao@acUSD.edu`

Abstract

This article reflects upon this author's unexpected, but most pleasant and productive experience teaching CS1, the first computer science programming class in the Spring semester of 1999, using Python as the primary language. It proposes that, no matter what your personal philosophy about computer science (CS) teaching is, you and your students can benefit from using Python in your classes.

1. Background

I taught CS1 for the first time in 1983 using Pascal. Pascal was a good teaching language, and the class went well. Since then, I have watched my department try different languages in CS1 including Modula-2, C++, Oberon, and Java. I did not teach CS1 again until the Spring semester of 1999. When that semester started, the department found itself without CS1 instructors. Since CS1 is such a fundamental course for CS majors and I am the chair of the Math/CS department, I volunteered to teach a section. Even though Java textbooks had already been ordered for all the CS1 sections, and even though I had been an ardent advocate of Java for CS1 instruction as it grew in popularity, I quickly decided that Java needed some help. While trying to find inspiration for my CS1 class, I chanced upon Mark Lutz's Python book[1] that had been gathering dust on my bookshelf. I decided to use Python as the primary language and Java as a supplementary/complimentary language. This means that, when appropriate, programs would be written with Python and Java side by side.

2. Goals

Like every CS1 instructor, I need to teach problem-solving and programming techniques effectively. I want to make every class fun and exciting. Furthermore, I want the course to be much more than an intellectually stimulating exercise. I want students to leave my class with a language they can and will use for a long time to help them automate things, to glue software pieces together, and/or to manage their websites, etc. I chose Python to achieve all these goals.

I believe Python is an excellent choice because

- it has a short learning curve and long staying power, much like Pascal. This is appreciated by everybody, but especially important for amateur programmers who do not program on a daily basis and tend to forget complicated language details.
- its many merits including simple syntax, clear semantics, dynamic and flexible object-oriented features, encourage programmers to continue to write new Python code and improve their existing Python code.

I also want the attrition be low.

3. Specifically, why is Python useful in teaching?

Python has the following good features that can help teaching. The examples given here focus on presentation clarity, not on code efficiency or good-looking output. Similar examples were presented in class by projecting the human-computer interaction onto a big screen.

1. *Simple, clean syntax and clear semantics.*

Common wisdom says that this lets the students concentrate on the problems they are solving, rather than on the language they are using.

The following example shows the psychologically satisfying *Hello World* programs written in Python and Java side-by-side, as presented in class.

Python code	Java code
-----	-----
<pre>print "Hello World!"</pre>	<pre>class HelloWorld{ public static void main(String[] args){ System.out.println("Hello World!"); } }</pre>

Python code takes little explanation. In contrast, Java code requires a lot of explanation to beginners. I tell my students to treat everything except

`System.out.println("Hello World!");` as packaging material, and to read `System.out.println` as an idiom for *print*.

2. *Interpreter can be used interactively. This makes it easy to learn and, most importantly, it encourages experimentation.*

To run the Python *Hello World* program, you can use the interactive Python interpreter. Here is a snapshot of a run on my Sun Sparc:

```
/home/liao/python [9]> python
```

```
Python 1.5.1 (#1, Jun 7 1998, 00:36:27) [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam1
>>> print "Hello World!"
Hello World!
>>>
```

Alternatively, we can store the Python program in a file, say, *hello.py*, and run it like this:

```
/home/liao/python [11]> cat hello.py      (just to see the program)
print "Hello World!"
/home/liao/python [12]> python hello.py
Hello World!
/home/liao/python [14]>
```

In contrast, to run the Java *Hello World* program, we need to store the program in a *Hello.java* file, compile it using *javac*, and then run it using *java*, like this:

```
/home/liao/java [16]> javac Hello.java
/home/liao/java [17]> java Hello
Hello World!
/home/liao/java [18]>
```

Beginners often either confuse *java* and *javac*, not sure when to use which, or carelessly type the wrong thing.

3. Control structures and functions can be covered easily.

In contrast, in Java, everything must be packaged in a class in order to execute. Therefore, to see control structures and functions in action, you need to put them in the context of a class. This introduces unnecessary complexities for beginners.

Some straightforward examples:

```
/home/liao/python [106]> cat towerOfHanoi.py
def solveHanoi(n,a,b):
    ''' this defines a recursive procedure to solve
        the problem of moving n disks from peg a
        to peg b, where a, b = 1, 2, 3 and are distinct '''
    c=6-(a+b)
    if n==2:
        print "move from peg ",a, " to peg ",c
        print "move from peg ",a, " to peg ",b
        print "move from peg ",c, " to peg ",b
```

1. These two lines of greeting will be omitted in the following.

```

else:
    solveHanoi(n-1,a,c)
    print "move from peg ",a, " to peg ",b
    solveHanoi(n-1,c,b)

/home/liao/python [81]> python
>>> from towerOfHanoi import *
>>> solveHanoi(4,1,3)
move from peg 1 to peg 2
move from peg 1 to peg 3
move from peg 2 to peg 3
move from peg 1 to peg 2
move from peg 3 to peg 1
move from peg 3 to peg 2
move from peg 1 to peg 2
move from peg 1 to peg 3
move from peg 2 to peg 3
move from peg 2 to peg 1
move from peg 3 to peg 1
move from peg 2 to peg 3
move from peg 1 to peg 2
move from peg 1 to peg 3
move from peg 2 to peg 3
>>>

```

4. Python debugger can be covered early on to be used as a learning tool.

To debug, I ask students to print out their programs and then read the program line by line to find errors. I advise them to use debugger only if they cannot find their errors after a few careful readings of their programs. But I think a debugger can be a useful learning tool. So, Python debugger was covered very early on. What follows are two examples of the use of *pdb*, the standard Python debugger in class. The first example traces the execution of an iterative function. The second example shows the call stack of the execution of a recursive function. Both functions find the greatest common divisor of two positive integers. They are both called *gcd*, with the iterative version defined in *gcd1.py* and the recursive one defined in *gcd2.py*.

```

/home/liao/python [115]> python
>>> (first example)
>>> from gcd1 import gcd
>>> import pdb # Python debugger is in the pdb library module

```

```

>>> pdb.run("gcd(15,20)")
> <string>(0)?()->None
(Pdb) s          # step into
> <string>(1)?()->None
(Pdb) s
> gcdl.py(4)gcd()
-> def gcd(m,n):
(Pdb) list
1 # gcdl.py
2
3 # assume m and n are both positive integers
4 ->def gcd(m,n):
5     while m<>n:
6         if m>n: m=m-n
7         else: n=n-m
8     return m
(Pdb) b 5          # set breakpoint at line 5
(Pdb) c          # continue
> gcdl.py(5)gcd()
-> while m<>n:
(Pdb) c
> gcdl.py(5)gcd()
-> while m<>n:
(Pdb) p m,n      # print m and n
(15, 20)
(Pdb) c
> gcdl.py(5)gcd()
-> while m<>n:
(Pdb) p m,n
(15, 5)
(Pdb) c
> gcdl.py(5)gcd()
-> while m<>n:
(Pdb) p m,n
(10, 5)
(Pdb) c

```

```

> gcd1.py(5)gcd()
-> while m<n:
(Pdb) p m,n
(5, 5)
(Pdb) c
--Return--
> <string>(1)?()->None
(Pdb) quit
>>> (second example)
>>> from gcd2 import gcd
>>> pdb.run("gcd(15,20)")
(Pdb) b gcd # set breakpoint on the function gcd
(Pdb) c
> <string>(1)?()->None
(Pdb) c
> gcd2.py(1)gcd()
-> def gcd(x,y):
(Pdb) c
> gcd2.py(1)gcd()
-> def gcd(x,y):
(Pdb) c
> gcd2.py(1)gcd()
-> def gcd(x,y):
(Pdb) c
> gcd2.py(1)gcd()
-> def gcd(x,y):
(Pdb) p x,y
(5, 5)
(Pdb) where # displays the call stack. The stack grows downwards.
<string>(1)?()->None
gcd2.py(4)gcd()
-> else: return gcd(x, y-x)
gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)
gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)

```

```

> gcd2.py(1)gcd()
-> def gcd(x,y):
(Pdb) up          # go up a frame
> gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)
(Pdb) p x,y
(10, 5)
(Pdb) u
> gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)
(Pdb) p x,y
(15, 5)
(Pdb) u
> gcd2.py(4)gcd()
-> else: return gcd(x, y-x)
(Pdb) p x,y
(15, 20)
(Pdb) u
> <string>(1)?()->None
(Pdb) down
> gcd2.py(4)gcd()
-> else: return gcd(x, y-x)
(Pdb) p x,y
(15, 20)
(Pdb) d
> gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)
(Pdb) p x,y
(15, 5)
(Pdb) d
> gcd2.py(3)gcd()
-> elif x>y: return gcd(x-y,y)
(Pdb) p x,y
(10, 5)
(Pdb) d
> gcd2.py(1)gcd()

```

```

-> def gcd(x,y):
(Pdb) p x,y
(5, 5)
(Pdb) d
*** Newest frame
(Pdb) c
--Return--
> <string>(1)?()->None
(Pdb) q
>>>

```

5. Python is object-oriented.

Beginners will find it easy to learn object orientation by doing experimentations in the Python interactive interpreter, like this:

```

/home/liao/python [19]> python
>>> # import the fractionModule module, which is shown below
...
>>> from fractionModule import *
>>> # next, I create a fraction 12/-45
...
>>> f = fraction(12,-45)
>>> f
12/-45
>>> f.simplify()
>>> f
4/-15
>>> g = fraction(45,12)
>>> g
45/12
>>> h = f.add(g)
>>> i = g.add(f)
>>> h
-627/-180
>>> i
-627/-180
>>> """ we now compute 1+(1/2)+(1/3)+ ... +(1/20) """
' we now compute 1+(1/2)+(1/3)+ ... +(1/20) '

```

```

>>> sum = fraction(0,1)
>>> sum
0/1
>>> for i in range(20): # range(20)=[0,1,2,...,17,18,19]
...     addend = fraction(1,i+1)
...     sum = sum.add(addend)
...
Traceback (innermost last):
  File "<stdin>", line 3, in ?
  File "fractionModule.py", line 26, in add
    denominator = self.denom*other.denom
OverflowError: integer multiplication
>>>

```

We ran into a problem for the last experiment. The error message tells where the error occurred. Let me show the source of *fractionModule.py*; then I will show how to *easily* fix this problem.

```

1  # fractionModule.py
2
3  """ this module defines a fraction class """
4
5  from gcd1 import *
6
7  class fraction:
8
9      def __init__(self,x,y):
10         """ constructor """
11         self.num = x
12         self.denom = y
13
14     def equal(self,other):
15         """ decide whether self and other have
16             the same values as fractions """
17         return self.num*other.denom == self.denom*other.num
18
19     def simplify(self):
20         """ simplify the fraction to the simplest terms """
21         if self.num == 0: self.denom=1

```

```

22     else:
23         quotient = gcd(abs(self.num), abs(self.denom))
24         self.num = self.num/quotient
25         self.denom = self.denom/quotient
26         if self.num<0 and self.denom<0:
27             self.num = -self.num
28             self.denom = -self.denom
29
30     def add(self,other):
31         """ return a fraction which is self + other """
32         denominator = self.denom*other.denom
33         numerator = self.num*other.denom + self.denom*other.num
34         theFraction = fraction(numerator, denominator)
35         return theFraction
36
37     def __repr__(self):
38         """ this gives a string representation of the fraction """
39         return `self.num` + `/'` + `self.denom`

```

The fraction.py requires the service of *gcd1.py*, which is shown next:

```

1  # gcd1.py
2
3  # assume m and n are both positive integers
4  def gcd(m,n):
5      while m<>n:
6          if m>n: m=m-n
7          else: n=n-m
8      return m

```

The integer overflow was caused by the computation

$$1*2*3*\dots*n$$

We can find out when overflow occurs using the Python interpreter, like this:

```

>>> p =1
>>> for i in range(20):
...     p = p*(i+1)
...     print "i=",i," ,", i+1 , "!=" ,p
...

```

```

i= 0 , 1 != 1
i= 1 , 2 != 2
i= 2 , 3 != 6
i= 3 , 4 != 24
i= 4 , 5 != 120
i= 5 , 6 != 720
i= 6 , 7 != 5040
i= 7 , 8 != 40320
i= 8 , 9 != 362880
i= 9 , 10 != 3628800
i= 10 , 11 != 39916800
i= 11 , 12 != 479001600
Traceback (innermost last):
  File "<stdin>", line 2, in ?
OverflowError: integer multiplication

```

This calls for long integers:

```

>>> p = 1L # p is a long integer
>>> for i in range(20):
...     p = p*(i+1)
...
>>> p
2432902008176640000L
>>>

```

We can solve the overflow problem easily by forcing the numerator, denominator and gcd all be long integers, which have infinite precision. This means to change line 8 of *gcd1.py* from

```
return m
```

to

```
return m*1L
```

In the same vein, change lines 11 and 12 of *fractionModule.py* from

```
self.num = x
self.denom = y
```

to

```
self.num = x * 1L
self.denom = y * 1L
```

and finally change line 21 of *fractionModule.py* from

```
    if self.num == 0: self.denom=1
to
```

```
    if self.num == 0: self.denom=1L
```

After this four-line change, the computation of $1+(1/2)+\dots+(1/19)+(1/20)$ will succeed:

```
>>> from fraction import *
>>> sum = fraction(0,1)
>>> for i in range(20):
...     addend = fraction(1,i+1)
...     sum = sum.add(addend)
...
>>> sum
8752948036761600000L/2432902008176640000L
>>>
```

6. *Python is free and can be easily downloaded[4] and installed on common platforms such as Mac, PC and machines running various flavors of Unix.*

This means that instructors do not need a budget to use Python in their classes. They should need very little assistance from the school's computing center.

7. *The existence of JPython[4] makes Python even more attractive because we can now access Java libraries from Python, and we can use Java libraries interactively from within a Python interactive interpreter. JPython is a Python interpreter written in 100% Java by Jim Hugunin.*

The following example shows the communication between a client socket and a server socket interactively using JPython. This is useful in a client/server programming class.

```
~/python[1053]jpython
JPython 1.1beta2 on java1.2
Copyright (C) 1997-1999 Corporation for National Research Initiatives
>>> import java.net
>>> t = java.net.Socket("beowulf.ucsd.edu",80) # http server
>>> s = java.net.Socket("www.ucla.edu",13)     # daytime server
>>> s
Socket[addr=www.ucla.edu/164.67.80.80,port=13,localport=6738]
>>> i = s.getInputStream()
>>> c = i.read()
>>> print c
83
```

```

>>> print chr(c)
S
>>> while c<>-1:
...     print chr(c),
...     c = i.read()
...
S u n   J u n   2 7   1 5 : 2 0 : 0 0   1 9 9 9
>>> i.close()
>>> s.close()
>>>

```

8. *Python has a rich set of libraries. It is a real, production language. It has a large, enthusiastic, active user group.*

This may not be crucial in CS1, but can be very attractive in other courses.

4. Anecdotal Stuff

4.1 I did not know Python

When I decided to use Python in my CS1 on Feb 1, 1999, I had just read a part of the Python tutorial from Mark Lutz's Python book[1]. That was all the Python that I knew. I learned Python quickly enough to not only excite my CS1 class, but to also write some useful programs that helped me do the department class scheduling¹. I would like to thank Guido van Rossum for creating such a wonderful language.

4.2 A demo of Guido's Tower of Hanoi program on three of my office machines excited the students

When we talked about the Tower of Hanoi problem, I ran Guido's program that comes with the Python distribution² on all of my office machines: a Windows PC³, a Sun Sparc, and a PC running RedHat Linux. I asked the students to view the demo before the class. They were excited, amused and inspired. Throughout the day, students kept on coming back to see the programs running with 10 disks.

1. One such program connects to a MySQL database that stores the department schedule and presents the schedule in various Web tables that make potential scheduling conflicts jump out and make it easier to schedule part-time teaching faculty. For a demo of this program, please see <http://beowulf.acusd.edu/~liao/99f.html>. The Python code is at <http://beowulf.acusd.edu/~liao/timeTable.py>.

2. in `Demo/tkinter/guido/hanoi.py`

3. running Python from a Linux machine.

4.3 The attrition was low and the class was enthusiastic

The class started with 26 students. Three quickly dropped in the first two weeks. The remaining 23 students stayed enthusiastic the whole time except one who stopped coming to class two weeks before the final. As of June 1999, twelve of them signed up for CS2 and/or more advanced CS courses for the Fall semester.

5. Some Problems

5.1 No CS1 textbooks

When I made the decision to use Python in my CS1 class on Feb 1, 1999, there were two Python books[1][3] in print. Both are beautifully written and wonderfully useful Python references, but neither is a good CS1 textbook. Some time during the semester, some students bought a newly published Python book[2]. This is another beautifully written and wonderfully useful Python reference, but it is still not a CS1 textbook. A search on amazon.com on Python on June 25, 1999 listed four new Python books that are due out soon, but none of which is a CS1 textbook.

Even though I ordered Mark Lutz's Python book[1] for my CS1 students, the book is not suitable as a textbook. In early February, a publisher wrote the following about Python:

Dear Professor Liao,

I recently got a chance to spend some time looking at the Python website. Thank you for sending me the URL. I'm surprised I hadn't heard of the language before, as many of the references on the website date back to 1996. I have been asking a number of computer science professors (primarily people who are our authors and reviewers, as well as professors I have met on my recent travels) if they have ever heard of Python. It seems most people haven't. Although it seems like it is a very interesting, and easy to use object-oriented language, I'm afraid I'm not going to be able to pursue your project at this time. There's just not enough recognition of Python in the academic market currently to support the publishing of a textbook on the subject. As I spend much of my time traveling to various campuses throughout the U.S., I will continue to ask people about Python. If I find, over time, more people are familiar with it, I will certainly re-contact you about your manuscript idea.

Even though the Python language manual and the Python library reference manuals[4] are all excellent, they do not substitute for CS1 textbooks. Textbooks written for Python will make instructors' lives easier.

5.2 Differences in terminologies and notations can be illuminating to some, but confusing to others

Very common terms can have widely different meanings in different languages, such as Python, Java and English. As an example, students think of chairs and balls when we first mention *objects* in class. They soon understand that objects in programming languages are different from objects in their daily languages. If they are diligent in reading Python and Java literature, they will be surprised that objects in different programming languages can mean drastically different things: Java objects are class instances; Python objects not only include class instances, but also modules, functions, and other data (including code) abstraction. As far as CS1 is concerned, we can safely de-emphasize their differences and restrict the discussion of Python objects to class instances.

Most of my students seemed to find Java class definitions easier to read and write, perhaps for these reasons:

- In Java, the class instance reference, called *this*, is hidden as the implicit first argument of a method. Its counterpart in Python, usually called *self*, must be the explicit first parameter of a method, and must be used explicitly in the method body to access the instance's members. Use lines 3 and 4 of the following Python class definition as an example: in line 3, *self* must be an explicit first parameter of the method *acctBalance()*; in line 4, the instance variable *balance* must be qualified as *self.balance*.

```
# Python class
class checkingAccount:
    def acctBalance(self):      (line 3)
        return self.balance    (line 4)
    def deposit(self,amt):
        if amt > 0:
            self.balance = self.balance + amt
        else:
            print "deposit failed"
    def withdraw(self,amt):
        if self.balance >= amt >0:
            self.balance = self.balance - amt
        else:
            print "withdraw failed"
    def __init__(self,initialbalance):
        self.balance = initialbalance
```

Compare this with an equivalent Java class definition:

```
// Java class
class CheckingAccount{
```

```

double balance;
double balance(){           (line 4)
    return balance;        (line 5)
}
void deposit(double amt){
    if (amt>0) {
        balance = balance + amt;
    } else {
        System.out.println("deposit failed");
    }
}
void withdraw(double amt){
    if ((amt>0) && (amt <= balance)){
        balance = balance - amt;
    } else {
        System.out.println("withdraw failed");
    }
}
CheckingAccount(double initialBalance){
    balance = initialBalance;
}
}

```

Lines 4 and 5 of the Java code show that the accessor method *balance()* does not have an explicit parameter, and the instance variable *balance* can be accessed without qualification.

- Java's enforced declarations of variables have the effects of clarifying the code. In Python, programmers must exercise their discipline to annotate variables to clarify their intents.

5.3 All attributes (variables or methods) of a class are effectively public.

There are no private or protected attributes as in Java. In the Python example in 5.2, the *balance* instance variable can be read or changed by a client of the class. To me, this is one more justification to use Java as a supplementary/complimentary language.

6. Conclusion

I have had a very positive experience teaching CS1 using Python as the primary language. I will report my experience using Python and JPython in a Client/Server Programming class this summer, and Web Programming class in the Fall, in separate articles.

7. Reference

- [1] Mark Lutz, *Programming Python*, ORA 1996
- [2] Mark Lutz & David Ascher, *Learning Python*, ORA 1999
- [3] Aaron Watters, Guido van Rossum & James Ahlstrom, *Internet Programming with Python*, M&T 1996
- [4] Python web site: <http://www.python.org>