Simple, not Simplistic
Squeezing the most from CS1
Python!

John M. Zelle, Ph.D.
Wartburg College

Background

- Teaching since 1986
- CS1 languages: Pascal, C++, Java (also CS0 BASIC)
- Favorite class but...
  increasingly frustrating
- Students stopped "getting it"
  - Student confusion, apathy, dropout
  - Inability to complete simple programs
  - Declining student evaluations
- Is it me?

Outline

- Motivation
- Introduction to Python
- Approaches to CS1
- Python Resources
- Conclusions
- Questions?

Rethinking CS1

- Learning Challenges
  - More material (software development, OOP, GUIs)
  - Complex Languages (systems languages Ada, C++, Java)
  - Complex Environments
  - Too much "magic"
- Teaching Challenges
  - Recruiting Majors
  - Serving Nonmajors
- Einstein: Make everything as simple as possible, but not simpler.

The March of Progress (Cay Horstmann)

- C | Pascal

```
printf("%10.2f", x); | write(x:10:2)
```

- C++

```
cout << setw(10) << setprecision(2)
     << showpoint << x;
```

- Java

```
java.text.NumberFormat formatter
   = java.text.NumberFormat.getNumberInstance();
formatter.setMinimumFractionDigits(2);
formatter.setMaximumFractionDigits(2);
String s = formatter.format(x);
for (int i = s.length(); i < 10; i++)
   System.out.print(' ');
System.out.print(s);
```

Why Use Python?

- Traditional languages (C++, Java) evolved for large-scale programming
  - Emphasis on structure and discipline
  - Simple problems != simple programs
- Scripting languages (Perl, Python, TCL) designed for simplicity and flexibility.
  - Simple problems = simple, elegant solutions
  - More amenable to experimentation and incremental development
- Python: Near ideal first language, useful throughout curriculum
- We've used it in CS1 since 1998

Enter Python

- Python: A free, portable, dynamically-typed, object-oriented scripting language
- Combines software engineering features of traditional systems languages with power and flexibility of scripting languages
- Real world language
- Batteries included
- Note: Named after Monty Python's Flying Circus

First Program (Java Version)

- Assignment: Print "Hello CCSC" on screen

```
public class Hello{
    public static void main(String [] args){
        System.out.println("Hello CCSC");
    }
}
```

- Note: Must be in "Hello.java"

First Program (Python Version)

- Assignment: Print "Hello CCSC" on screen

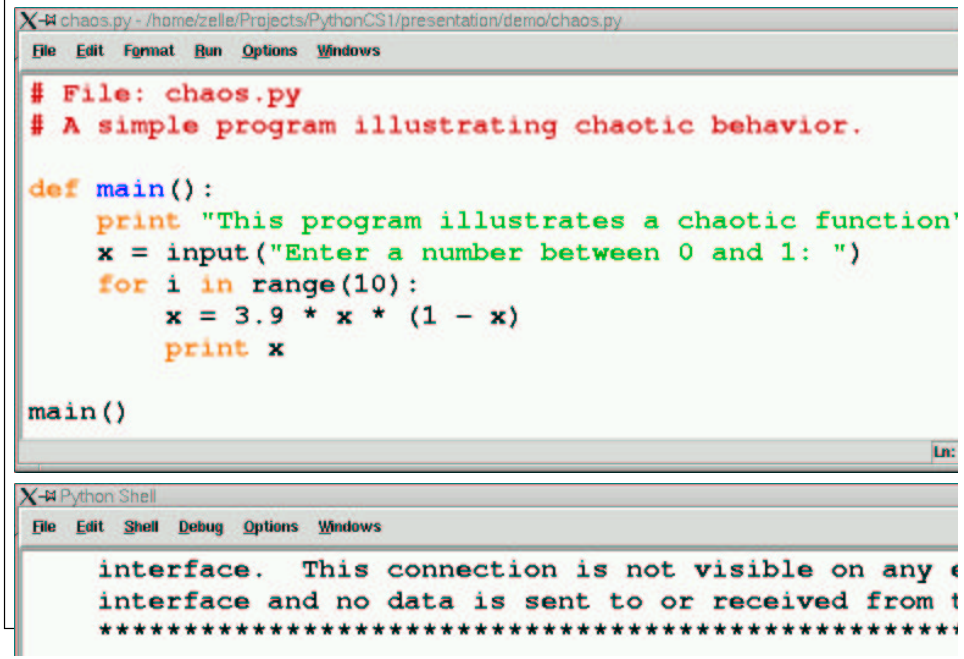
```
print "Hello CCSC"
```

- Or...

```
def main():
    print "Hello CCSC"

main()
```

Example in IDLE



"Real" Program: Chaos.py

```
#File: chaos.py
# A simple program illustrating chaotic behavior.

def main():
   print "This program illustrates a chaotic function"
   x = input("Enter a number between 0 and 1: ")
   for i in range(10):
       x = 3.9 * x * (1 - x)
       print x

main()
```

Basic Statements

- Output

  ```
  print <expr1>, <expr2>, ..., <exprn>
  ```

  - Note: all Python types have printable representations

- Simple Assignment

  ```
  <var> = <expr>
  myVar = oldValue * foo + skip
  ```

- Simultaneous Assignment

  ```
  <var1>, <var2>, ... = <expr1>, <expr2>, ...
  a,b = b,a
  ```

- Assigning Input

  ```
  input(<prompt>)
  myVar = input("Enter a number: ")
  x,y = input("Enter the coordinates (x,y): ")
  ```

Example Program: Fibonacci

```
# fibonacci.py
# This program computes the nth Fibonacci number

n = input("Enter value of n ")

cur,prev = 1,1
for i in range(n-2):
    cur,prev = prev+cur,cur

print "The nth Fibonacci number is", cur
```

Teaching Tip: Indentation as Syntax

- Pluses
  - less code clutter (; and {})
  - eliminates most common syntax errors
  - promotes and teaches proper code layout
- Minuses
  - occasional subtle error from inconsistent spacing
  - will want an indentation-aware editor
- Bottom-line: Good Python editors abound.
  This is my favorite feature.

Teaching Tip: Dynamic Typing

- Pluses
  - less code
  - less upfront explanation
  - eliminates accidental redeclaration errors
- Minuses
  - typo on LHS of = creates new variable
  - allows variables to change type
- Bottom-line: I prefer dynamic types
  - Many (most?) type errors are declaration errors
  - Actual type errors are still detected
  - Finding type errors goes hand-in-hand with testing
  - Less student frustration

Numeric Types

- int: Standard 32 bit integer
  `32 -3432 0`
- long int: Indefinitely long integers
  `32L 9999999999999999`
- floating-point: Standard double-precision float
  `3.14  2.57e-10  5E210  -3.64e+210`
- complex: Double precision real and imaginary components
  `2+3j  4.7J  -3.5 + 4.3e-4j`
- User-defined types (operator overloading)

Numeric Operations

- Builtins
  +, -, *, /, %, **, abs(), round()
- Math Library
  pi, e, sin(), cos(), tan(), log(), log10(), ceil(), ...

String Datatype

- String is an immutable sequence of characters
- Literal delimited by ’ or " or """

```
s1 = ’This is a string’
s2 = "This is another"
s3 = "that’s one alright"
s4 = """This is a long string that
goes across multiple lines.
It will have embedded end of lines"""
```

- Strings are indexed
  - From the left starting at 0 or...
  - From the right using negative indexes
- A character is just a string of length 1

Example Numeric Program: quadratic.py

```
# quadratic.py
# Program to calculate real roots
#    of a quadratic equation

import math

a, b, c = input("Enter the coefficients (a, b, c): ")

discRoot = math.sqrt(b * b - 4 * a * c)
root1 = (-b + discRoot) / (2 * a)
root2 = (-b - discRoot) / (2 * a)

print "\nThe solutions are:", root1, root2
```

String Operations

```
>>>"Hello, " + " world!"
’Hello, world!’

>>> "Hello" * 3
’HelloHelloHello’

>>> greet = "Hello John"
>>> print greet[0], greet[2], greet[4]
H l o

>>> greet[4:9]
’o Joh’
>>> greet[:5]
’Hello’
>>> greet[6:]
’John’

>>> len(greet)
10
```

Example Program: Month Abbreviation

```
months = "JanFebMarAprMayJunJulAugSepOctNovDec"

n = input("Enter a month number (1-12): ")
pos = (n-1)*3
monthAbbrev = months[pos:pos+3]

print "The month abbreviation is", monthAbbrev+"."
```

Standard String Library (string)

```
capitalize(s)  -- upper case first letter
capwords(s)    -- upper case each word
upper(s)       -- upper case every letter
lower(s)       -- lower case every letter

ljust(s, width)   -- left justify in width
center(s, width)  -- center in width
rjust(s, width)   -- right justify in width

count(substring, s)  -- count occurrences
find(s, substring)   -- find first occurrence
rfind(s, substring)  -- find from right end
replace(s, old, new) -- replace first occurrence

strip(s)  -- remove whitespace on both ends
rstrip(s) -- remove whitespace from end
lstrip(s) -- remove whitespace from front

split(s, char)   -- split into list of substrings
join(stringList) -- concatenate list into string
```

More String Operations

- Interactive input

```
s = raw_input("Enter your name: ")
```

- Looping through a string

```
for ch in name:
    print ch
```

- Type conversion
  - to string

```
>>> str(10)
'10'
```

  - from string

```
>>> eval('10')
10
>>> eval('3 + 4 * 7')
31
```

Example Programs: Text/ASCII Conversion

```
# Converting from text to ASCII codes
message = raw_input("Enter message to encode: ")

print "ASCII Codes:"
for ch in message:
    print ord(ch),

# Converting from ASCII codes to text
import string

inString = raw_input("Enter ASCII codes: ")

message = ""
for numStr in string.split(inString):
    message += chr(eval(numStr))

print "Decoded message:", message
```

String Formatting

- % operator inserts values into a template string (ala C printf)

```
<template-string> % (<values>)
```

- "Slots" specify width, precision, and type of value

```
%<width>.<precision><type-character>
```

- Examples

```
>>> "Hello %s %s, you owe %d" % ("Mr.", "X", 10000)
'Hello Mr. X, you owe 10000'

>>> "ans = %8.3f" % 3.14159265
'ans =    3.142'

print "%10.2f" % x  # apparently, a throwback :-)
```

File Processing

- Opening a file

```
syntax:  <filevar> = open(<name>, <mode>)
example: infile = open("numbers.dat", "r")
```

- Reading from file

```
syntax:  <filevar>.read()
         <filevar>.readline()
         <filevar>.readlines()
example: data = infile.read()
```

- Writing to file

```
syntax:  <filevar>.write(<string>)
example: outfile.write(data)
```

Example Program: Username Creation

- Usernames are first initial and 7 chars of lastname (e.g. jzelle).

```
inf = open("names.dat", "r")
outf = open("logins.txt", "w")

for line in inf:
    first, last = line.split()
    uname = (first[0]+last[:7]).lower()
    outf.write(uname+'\n')

inf.close()
outf.close()
```

- Note use of string methods (Python 2.0 and newer)

Functions

- Example:

```
def distance(x1, y1, x2, y2):
    # Returns dist from pt (x1,y1) to pt (x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx*dx + dy*dy)
```

- Notes:
  - Parameters are passed by value
  - Can return multiple values
  - Function with no return statement returns None
  - Allows Default values
  - Allows Keyword arguments
  - Allows variable number of arguments

Teaching Tip: Uniform Memory Model

- Python has a single data model
  - All values are objects (even primitive numbers)
  - Heap allocation with garbage collection
  - Assignment always stores a reference
  - None is a special object (analogous to null)
- Pluses
  - All assignments are exactly the same
  - Parameter passing is just assignment
- Minuses
  - Need to be aware of aliasing when objects are mutable

Booleans in Python

- Traditional Python: Conditions return 0 or 1 (for false, true)
- As of Python 2.3 bool type: True, False
- All Python built-in types can be used in Boolean exprs
  - numbers: 0 is False anything else is true
  - string: empty string is False, any other is true
  - None: False
- Boolean operators: and, or, not (short circuit, operational)

Decisions

```
if temp > 90:
    print "It's hot!"

if x <= 0:
    print "negative"
else:
    print "nonnegative"

if x > 8:
    print "Excellent"
elif x >= 6:
    print "Good"
elif x >= 4:
    print "Fair"
elif x >= 2:
    print "OK"
else:
    print "Poor"
```

Loops

- For loop iterates over a sequence

```
for <variable> in <sequence>:
    <body>
```

  - sequences can be strings, lists, tuples, files, also user-defined classes
  - range function produces a numeric list
  - xrange function produces a lazy sequence
- Indefinite loops use while

```
while <condition>:
    <body>
```

- Both loops support break and continue

Lists: Dynamic Arrays

- Python lists are similar to vectors in Java
  - dynamically sized
  - indexed (0..n-1) sequences
- But better..
  - Heterogeneous
  - Built into language (literals [])
  - Rich set of builtin operations and methods

List are Mutable

```
>>> x = [1, 2, 3, 4]

>>> x[1] = 5
>>> x
[1, 5, 3, 4]

>>> x[1:3] = [6,7,8]
>>> x
[1, 6, 7, 8, 4]

>>> del x[2:4]
>>> x
[1, 6, 4]
```

Sequence Operations on Lists

```
>>> x = [1, "Spam", 4, "U"]
>>> len(x)
4

>>> x[3]
'U'

>>> x[1:3]
['Spam', 4]

>>> x + x
[1, 'Spam', 4, 'U', 1, 'Spam', 4, 'U']

>>> x * 2
[1, 'Spam', 4, 'U', 1, 'Spam', 4, 'U']

>>> for i in x: print i,
1 Spam 4 U
```

List Methods

```
myList.append(x)   -- Add x to end of myList
myList.sort()      -- Sort myList in ascending order
myList.reverse()   -- Reverse myList
myList.index(s)    -- Returns position of first x
myList.insert(i,x) -- Insert x at position i
myList.count(x)    -- Returns count of x
myList.remove(x)   -- Deletes first occurrence of x
myList.pop(i)      -- Deletes and return ith element

x in myList        -- Membership check (sequences)
```

Example Program: Averaging a List

```
def getNums():
    nums = []
    while True:
        xStr = raw_input("Enter a number: ")
        if xStr == "": break
        nums.append(eval(xStr))
    return nums

def average(lst):
    sum = 0.0
    for num in lst:
        sum += num
    return sum / len(lst)

data = getNums()
print "Average =", average(data)
```

Dictionaries: General Mapping

- Dictionaries are a built-in type for key-value pairs (aka hashtable)
- Syntax similar to list indexing
- Rich set of builtin operations
- Very efficient implementation

Tuples: Immutable Sequences

- Python provides an immutable sequence called tuple
- Similar to list but:
  - literals listed in () Aside: singleton (3,)
  - only sequence operations apply (+, *, len, in, iteration)
  - more efficient in some cases
- Tuples (and lists) are transparently "unpacked"

```
>>> p1 = (3,4)
>>> x1, y1 = p1
>>> x1
3
>>> y1
4
```

Basic Dictionary Operations

```
>>> dict = { 'Python': 'Van Rossum', 'C++':'Stroustrup',
'Java':'Gosling'}

>>> dict['Python']
'Van Rossum'

>>> dict['Pascal'] = 'Wirth'

>>> dict.keys()
['Python', 'Pascal', 'Java', 'C++']

>>> dict.values()
['Van Rossum', 'Wirth', 'Gosling', 'Stroustrup']

>>> dict.items()
[('Python', 'Van Rossum'), ('Pascal', 'Wirth'), ('Java',
'Gosling'), ('C++', 'Stroustrup')]
```

More Dictionary Operations

```
del dict[k]          -- removes entry for k
dict.clear()         -- removes all entries
dict.update(dict2)   -- merges dict2 into dict
dict.has_key(k)      -- membership check for k
k in dict            -- Ditto
dict.get(k,d)        -- dict[k] returns d on failure
dict.setDefault(k,d) -- Ditto, also sets dict[k] to d
```

Python Modules

- A module can be:
  - any valid source (.py) file
  - a compiled C or C++ file
- A single module can contain any number of structures
  - Example: graphics.py (GraphWin, Point, Line, Circle, color_rgb,...)
- Locating modules
  - Default search path includes Python lib and current directory
  - Can be modified when Python starts or by program (sys.path)
  - No naming or location restrictions
- Also supports directory structured packages

```
from OpenGL.GL import *
from OpenGL.GLUT import *
```

Example Program: Most Frequent Words

```
import string, sys

text = open(sys.argv[1],'r').read()
text = text.lower()
for ch in string.punctuation:
    text = text.replace(ch, ' ')

counts = {}
for w in text.split():
    counts[w] = counts.get(w,0) + 1

items = [(c,w) for (w,c) in counts.items()]
items.sort()
items.reverse()

for c,w in items[:10]:
    print w, c
```

Teaching Tip: Information Hiding

- In Python, Information hiding is by convention
  - All objects declared in a module can be accessed by importers
  - Names beginning with _ are not copied over in a from...import *
- Pluses
  - Makes independent testing of modules easier
  - Eliminates visibility constraints (public, protected, private, static, etc.)
- Minuses
  - Language does not enforce the discipline
- Bottom-line: Teaching the conventions is easier
  - The concept is introduced when students are ready for it
  - Simply saying "don't do that" is sufficient (when grades are involved).

Python Classes: Quick Overview

- Objects in Python are class based (ala SmallTalk, C++, Java)
- Class definition similar to Java

```
class <name>:
    <method and class variable definitions>
```

- Class defines a namespace, but not a classic variable scope
  - Instance variables qualified by an object reference
  - Class variables qualified by a class or object reference
- Multiple Inheritance Allowed

Using a Class

```
>>> from msdie import *
>>> d1 = MSDie(6)
>>> d1.roll()
>>> d1.getValue()
6
>>> d1.roll()
>>> d1.getValue()
5
>>> d1.instances
1
>>> MSDie.instances
1
>>> d2 = MSDie(13)
>>> d2.roll()
>>> d2.value
7
>>> MSDie.instances
2
```

Example: a generic multi-sided die

```
from random import randrange

class MSDie:

    instances = 0   # Example class variable

    def __init__(self, sides):
        self.sides = sides
        self.value = 1
        MSDie.instances += 1

    def roll(self):
      self.value = randrange(1, self.sides+1)

    def getValue(self):
        return self.value
```

Example with Inheritance

```
class SettableDie(MSDie):

    def setValue(self, value):
        self.value = value

-------------------------------------------
>>> import sdie
>>> s = sdie.SettableDie(6)
>>> s.value
1
>>> s.setValue(4)
>>> s.value
4
>>> s.instances
3
```

Notes on Classes

- Data hiding is by convention
- Namespaces are inspectable

```
>>> dir(sdie.SettableDie)
['__doc__', '__init__', '__module__', 'getValue',
'instances', 'roll', 'setValue']
>>> dir(s)
['__doc__', '__init__', '__module__', 'getValue',
'instances', 'roll', 'setValue', 'sides', 'value']
```

- Attributes starting with __ are "mangled"
- Attributes starting and ending with __ are special hooks

Another Class: Just for Fun

```
 #file: stack.py

"""Implementation of a classic
stack data structure: class Stack"""

class Stack:

    "Stack implements a classic stack with lists"

    def __init__(self): self.data = []

    def push(self, x): self.data.append(x)

    def top(self): return self.data[-1]

    def pop(self): return self.data.pop()
```

Documentation Strings (Docstrings)

- Special attribute __doc__ in modules, classes and functions
- Python libraries are well documented

```
>>> from random import randrange
>>> print randrange.__doc__
Choose a random item from range(start, stop[, step]).

        This fixes the problem with randint() which includes the
        endpoint; in Python this is usually not what you want.
        Do not supply the 'int' and 'default' arguments.
```

- Used by interactive help utility

```
>>> help(randrange)
$ pydoc random.randrange
```

- Docstrings are easily embedded into new code
  - can provide testing framework

Advantages for CS1

- Simple language = More time for concepts
- Safe loop and rich built-ins = Interesting programs early
- Free Language and IDE = Easy for students to acquire
- Dynamic features = Ease of experimentation
- Less code = More programming assignments

Our Approach

- Spiral of imperative and OO concepts (objects ontime?)
- Emphasize:
  - Algorithmic thinking
  - Universal design/programming patterns (not Python)
- Outline
  - Simple numeric processing first
  - String processing by analogy to numeric
  - Using objects via graphics
  - Functions and control structures
  - Top-down design
  - Classes
  - Collections
  - OO Design
  - Algorithm Design and Recursion

Graphics Example: triangle.py

```
from graphics import *  # our custom graphics

win = GraphWin("Draw a Triangle")
win.setCoords(0.0, 0.0, 10.0, 10.0)
message = Text(Point(5, 0.5), "Click on three points")
message.draw(win)
p1 = win.getMouse()
p1.draw(win)
p2 = win.getMouse()
p2.draw(win)
p3 = win.getMouse()
p3.draw(win)
triangle = Polygon(p1,p2,p3)
triangle.setFill("peachpuff")
triangle.setOutline("cyan")
triangle.draw(win)
message.setText("Click anywhere to quit.")
win.getMouse()
```

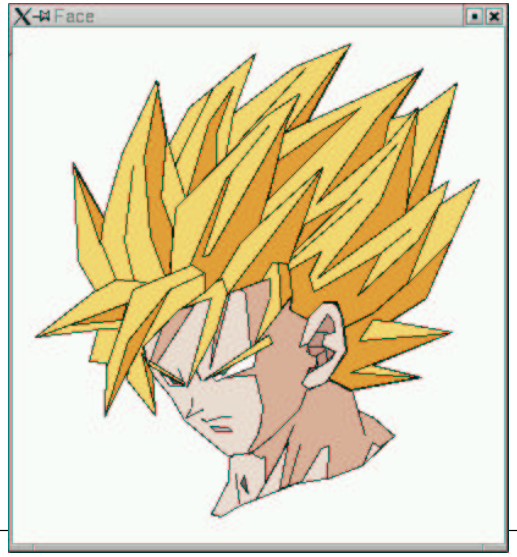
Graphics Library

- Homegrown 2D graphics package (graphics.py)
- Thin wrapper over Python standard GUI package Tkinter
- Why?
  - Students LOVE graphics, but it adds complexity
  - Our package "hides" the event loop
  - Teaches graphics and object concepts
- Natural progression
  - Learn by using concrete objects
  - Build own widgets
  - Implement simple event loop

Graphics Example: Triangle Screenshot

Graphics Example: Face

- Assignment: Draw something with a face



Graphics Example: Blackjack Project

Other Approaches to CS1

- Objects First
  - Rich set of readily useable objects
- Multi-Paradigm
  - Peter Norvig: '...a dialect of LISP with "traditional" syntax.'
- Breadth-First
  - perfect for first brush of programming
- 3D Graphics
  - VPython -- visualization for mere mortals
- GUI/Events early
  - Tkinter is (arguably) the simplest GUI toolkit going

What About CS2?

- Currently we use Java in CS2
- Why?
  - Want our students to see static typing
  - Java is a high-demand language
  - Switching languages is good for them
- It works
  - Students are better programmers coming in
  - The conceptual base is the same
  - They find Java annoying, but not difficult
  - Python is our pseudo-code
- My experience
  - CS2 is at least as smooth as before
  - Upper-level classes much better

Python Resources

- Textbooks (CS1, CS2)
  - "Python: How to Program," Deitel, Deitel, Liperi, Weidermann, and Liperi, (Prentice Hall)
  - "How to Think Like a Computer Scientist: Learning with Python," Downey, Elkner, and Meyers (Green Tea Press)
  - "Python Programming: An Introduction to Computer Science," Zelle (Franklin, Beedle, and Associates)
- Technical Python Books
  - Too many to list, see Python web site and Amazon
  - Personal Favorite: "Python in a Nutshell," Alex Martelli (O'Reilly and Assoc.)
- Python Web Sites
  - www.python.org -- The site for everything Pythonic
  - www.vex.net/parnassus/ -- Searchable database of Python add-ons

Conclusions

Python Rocks!

You'll Never Go Back