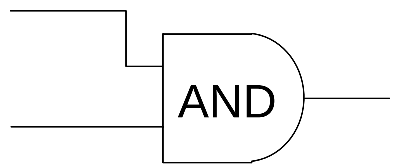
SIGCSE 2008

Combining
Discrete Mathematics

Python Programming



Maria Litvin
Phillips Academy
Andover, Massachusetts
mlitvin@andover.edu

Math and computer science should help each other:

- A programmer needs to be comfortable with abstractions, and that is precisely what math teaches.
- Computer science reciprocates by providing models and hands-on exercises that help clarify and illustrate more abstract math.
- Most importantly, both teach “precision thinking” — an important means of solving problems that call for exact solutions.

Slide 2

Why Python?

- Easy to install and get started with; has a simple built-in editor
- Has a convenient subset for novices
- Has straightforward syntax
- Provides easy console I/O and file handling
- Has simple but powerful features for working with strings, lists, “dictionaries” (maps), etc.
- Free

Slide 3

Lab 1: Sums and Iterations (from Ch 4)

$$1+2+...+n=\frac{n(n+1)}{2}$$

Proof:

$$\begin{array}{lcccccccc} & s= & \boxed{1} & + & \boxed{2} & + ... & + & \boxed{n} \\ + & & + & & + & & & + \\ & s= & n & + & (n-1) & + ... & + & 1 \\ \hline \end{array}$$

$$2s=\underbrace{(n+1)+(n+1)+...+(n+1)}_{n \text{ times}}$$

$$2s=n(n+1) \Rightarrow s=\frac{n(n+1)}{2}$$

Slide 4

```
def sum1ToN(n):
    "Calculates 1+2+...+n using a formula"
    return n*(n+1)/2

n = -1
while n <= 0:
    s = raw_input("Enter a positive integer: ")
    try:
        n = int(s)
    except ValueError:
        print "Invalid input"

print 'n =', n
print '1+2+...+n =', sum1ToN(n)
```

Slide 5

```
def sum1ToN(n):
    "Calculates 1+2+...+n using a formula"
    return n*(n+1)/2

n = -1
while n <= 0:
    s = raw_input("Enter a positive integer: ")
    try:
        n = int(s)
    except ValueError:
        print "Invalid input"

print 'n =', n
print '1+2+...+n =', sum1ToN(n)
```

Define a function

Optional “documentation string”

Read a line from input

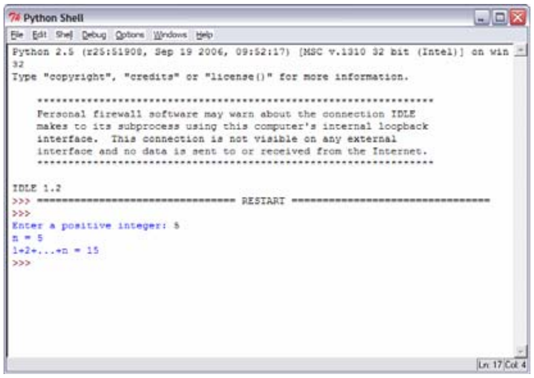
Single or double quotes can be used in literal strings

Slide 6

Lab 1 (cont’d)

- Run IDLE (Python’s GUI shell).
- Open a new editor window (File/New Window or Ctrl-N).
- Type in the program from Slide 6.
- Save the file (File/Save or Ctrl-S) as, say, Lab1.py in a folder (for example, in C:\sigcse2008-17).
- Run the program (Run/Run Module or F5).

Slide 7



Lab 1 (cont'd)

Now let's pretend that we do not know the formula for 1 + 2 + ... + *n* and calculate this sum using iterations.

Add the code from the next slide at the end of the same Python source file.


```
...

def sum1ToN(n):
   "Computes 1+2+...+n using iterations"
   s = 0
   k = 1
   while k <= n:
      s += k          # Short for  s = s + k
      k += 1
   return s

print '1+2+...+n =', sum1ToN(n)
```



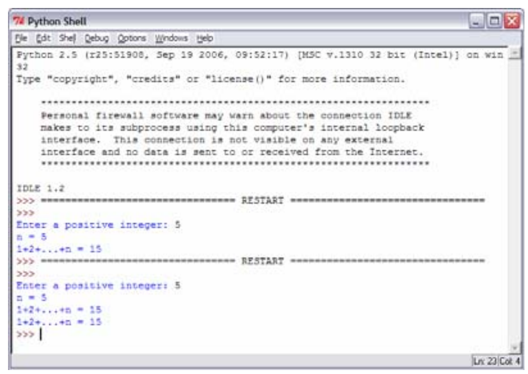
```
...

def sum1ToN(n):
   "Computes 1+2+...+n using iterations"
   s = 0
   k = 1
   while k <= n:
      s += k          # Short for  s = s + k
      k += 1
   return s

print '1+2+...+n =', sum1ToN(n)
```

Overrides the earlier definition

The global variable *n* is defined earlier (see Slide 6)



Lab 1 (cont'd)

Once you have run a program, its functions and global variables become "imported," and you can work with them interactively in Python shell. For example:

```
>>> n
5
>>> sum1ToN(5)
15
>>> sum1ToN(100)
5050
```

Exercise

Write a program that prompts the user to enter a positive integer *n* and displays the table

```
   1:     1      1
   2:     3      9
 ...
   n:  s1(n)  s3(n)
```

where

$s_1(n) = 1+2+...+n$

$s_3(n) = 1^3+2^3+...+n^3$

Guess the formula for $s_3(n)$.

See hints on the next slide.

Slide 14

Hints

- print "%3d: %3d %5d" % (k, s1, s3) prints k, s1, and s3 aligned in columns (the supported formats are similar to printf in C++, Java).
- Your program will be more efficient if you use only one while loop and update the values of s1, s3 on each iteration, instead of recalculating them each time from scratch. So do not use function calls.

Slide 15

Exercise "Extra Credit"

Change your program to also display $s_2(k)$ and $3 \cdot s_2(k) / s_1(k)$, where

$s_2(n) = 1^2+2^2+...+n^2$

Guess the formula for $s_2(n)$.

Slide 16

Lab 2: The Fundamental Theorem of Arithmetic (from Ch 15)

The fundamental theorem of arithmetic states that any positive integer can be represented as a product of primes and that such a representation is unique (up to the order of the factors). For example:

$90 = 2 \cdot 3 \cdot 3 \cdot 5$

The proof requires some work – it is not trivial.

Slide 17

Exercise

Write a program that prompts the user to enter a positive integer *n* and displays its prime factors separated by *. For example:

```
Enter a positive integer: 90
90 = 2 * 3 * 3 * 5
```

See hints on the next two slides.

Slide 18

Hints

- No need to look for primes – just take the smallest divisor *d* of *n* ($d > 1$), print it out, then divide *n* by *d*, and continue.
- The if statement

```
if n % d == 0:
    ...
else:
    ...
```

checks whether d divides n.

Slide 19

Hints (cont'd)

- One way to display asterisks correctly between the factors:

```
separator = '='
while ...:
    if ...:
        print separator, d,
        separator = '*'
```

This comma prevents newline -- the output will go to the same line

Exercise "Extra Credit"

Write and test a function that takes a positive integer and returns a string of its binary digits. For example, binDigits(23) should return '10111'.

Hints:

- str(d) converts a number d into a string
- Python's integer division operator is //
  (it truncates the result to an integer)
- s1 + s2 concatenates strings s1 and s2

Lab 3: Polynomials and Binomial Coefficients (from Ch 11)

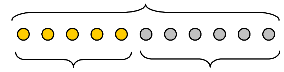
$\binom{n}{k}$ (read "n choose k") represents the number of ways in which we can choose *k* different objects out of *n* (where the order of the selected objects does not matter). For example, there are 108,043,253,365,600 ways to choose 23 workshop participants out of 50 applicants.

Lab 3 (cont'd)

Proof: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

Our selection method is to arrange *n* objects in a line, then take the first *k* of them.

n objects;
n! permutations



k objects;
k! permutations

n-k objects;
(n-k)!
permutations

Lab 3 (cont'd)

Factorials can get pretty big quickly, but Python automatically switches to large integers. For example:

```
>>> factorial(100)
9332621544394415268169
9238856266700490715968
2643816214685929638952175
9999322991560894146397
6156518286253697920827
2237582511852109168640000
00000000000000000000L
>>>
```

```
def factorial(n):
    f = 1
    k = 2
    while k <= n:
        f *= k
        k +=1
    return f
```


Lab 3 (cont'd)

Still, it is more efficient to avoid huge numbers. We can calculate n-choose-k using the following property:

$$\binom{n}{k} = \binom{n}{k-1} \cdot \frac{n-k+1}{k}$$

```
def nChooseK(n, k):  # recursive version
    if k == 0:
        return 1
    else:
        return nChooseK(n, k-1) * (n - k + 1) / k
```

Lab 3 (cont'd)

The n-choose-k numbers are also known as *binomial coefficients* because

$$(x+1)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1} + \ldots + \binom{n}{n-1}x + \binom{n}{n}$$

So we can compute n-choose-k by multiplying polynomials (and in the process get a feel for handling lists in Python).

Lists in Python

```
lst1 = [2, 3, 5, 7, 11]
len(lst1)              # 5
i = 3
lst1 [ i ]             # 7
lst2 = lst1[1:3]       # a "slice" of lst1: [3, 5]
lst1a = lst1[:]        # a copy of lst1
lst0 = [ ]             # an empty list
lst3 = 3*lst2          # [3, 5, 3, 5, 3, 5]
lst1.append(13)        # [2, 3, 5, 7, 11, 13]
lst4 = lst1 + [17, 19] # [2, 3, 5, 7, 11, 13, 17, 19]
lst5 = 5*[0]           # [0, 0, 0, 0, 0]
```


Lab 3 (cont'd)

Let's represent a polynomial

$$a_n x^n + \ldots + a_1 x + a_0$$

as a list of its coefficients

$[a_n, \ldots, a_1, a_0]$

The function multiply(p1, p2) returns the product of two polynomials (represented as a list).

Lab 3 (cont'd)

```
def multiply(p1, p2):
  n = len(p1) + len(p2) - 1
  result = n*[0]

  i = 0
  while i < len(p1):
    j = 0
    while j < len(p2):
      result [ i+j ] += p1[ i ] * p2[ j ]
      j += 1
    i += 1

  return result
```

Length of the resulting list

Creates a list of length *n* filled with zeros

Indices start from 0, as usual

Exercise

Write a program that prompts the user for a positive integer *n* and prints a Pascal's triangle with *n* rows:

```
0: [1]
1: [1, 1]
2: [1, 2, 1]
3: [1, 3, 3, 1]
4: [1, 4, 6, 4, 1]
```

See hints on the next slide.

Hints

- The above code for the multiply function is available in Polynomials.py. Cut and paste or copy this file to your work folder and add

  from polynomials import multiply

  to your program.
- The polynomial x+1 is represented as [1, 1]
- Use print str(k) + ':' to print k followed by a colon
- Use print lst to print the list lst

Exercise "Extra Credit"

Add to the output for each row the sum of all the elements in that row and the sum of their squares. Show that

$$\binom{n}{0}^2+\binom{n}{1}^2+\ldots+\binom{n}{n}^2=\binom{2n}{n}$$

Proof: compare the middle coefficients in $(x+1)^{2n}$ and $(x+1)^n\cdot(x+1)^n$

See programming hints on the next slide.

Slide 32

Hints

- The built-in function sum(lst) returns the sum of the elements of lst.
- Python has a neat feature called "list comprehensions." For example, to obtain a list lst2 of squares of the elements of lst you can use a "list comprehension"

```
lst2 = [x*x for x in lst]
```

Slide 33

Lab 4: Probability of Matching Birthdays (from Ch 12)

What is the probability $p(k)$ that in a group of k people at least two have the same birthday?

$$p(k)=1-q(k)$$

where $q(k)$ is the probability that all the birthdays are different

$$q(k)=\frac{365\cdot364\cdot\ldots\cdot(365-k+1)}{365^k}$$

Slide 34

Exercise

Write a program that prints a table of pairs k, $p(k)$ for k from 1 to 50. Find the smallest k such that $p(k) > 0.5$.

Slide 35

Hints

- The program is just a few lines of code because

$$q(k+1)=q(k)\frac{365-k}{365}$$

- Be careful with division. Work with floating point numbers (e.g., 365.0, not 365) to avoid truncation in integer division or put

```
from __future__ import division
```

at the top of your program.

Slide 36

Back to the Big Picture...

- Math-in-CS debates notwithstanding, knowing relevant math makes better CS students and professionals.
- Start in middle school.
- "Problem solving" means solving problems, not just applying familiar skills in familiar ways.
- Proofs are not just boring exercises in geometry.
- Math+CS blend can bring new kinds of recruits to CS: young people who like math but have not considered CS.

Slide 37