

Essential Computing for Bioinformatics

Lecture 4

High-level Programming with Python

Part I: Controlling the flow of your program

Bienvenido Vélez

UPR Mayaguez

Reference: How to Think Like a Computer Scientist: Learning with Python (Ch 3-6)

Outline

- Functions
- Decision statements
- Iteration statements
- Recursion

Built-in Functions

```
>>> import math
>>> decibel = math.log10 (17.0)
>>> angle = 1.5
>>> height = math.sin(angle)
>>> degrees = 45
>>> angle = degrees * 2 * math.pi / 360.0
>>> math.sin(angle)
0.707106781187
```

To convert from degrees to radians,
divide by 360 and multiply by 2*pi

Can you avoid having to write the formula to
convert degrees to radians every time?

Defining Your Own Functions

```
def <NAME> ( <LIST OF PARAMETERS> ):  
    <STATEMENTS>
```

```
import math  
def radians(degrees):  
    result = degrees * 2 * math.pi / 360.0  
    return(result)
```

```
>>> def radians(degrees):  
...     result=degrees * 2 * math.pi / 360.0  
...     return(result)  
...  
  
>>> radians(45)  
0.78539816339744828  
>>> radians(180)  
3.1415926535897931
```

Monolithic Code

```
From string import *
```

```
cds = "atgagtgaacgtctgagcattaccccgctggggccgtatatcggcgcaaaa  
ttcgggtgccgacctgacgcgcccgttaagcgataatcagttgaacagctttaccatgcggtg  
ctgcgccatcaggtggtgtttctacgcgatcaagctattacgccgcagcagcaacgcgcgctggc  
ccagcgtttggcgaattgcatattaccctgtttaccgcatgccgaagggggtgacgagatca  
tcgtgctggatacccataacgataatccgccagataacgacaactggcataccgatgtgacattt  
attgaaacgccaccgcaggggcgattctggcagctaaagagttacctcgaccggcggtgatac  
gctctggaccagcggtattgcggcctatgaggcgctctctgttcccttccgccagctgctgagt  
ggctgctgaggagcatgattccgtaaatacgttcccggaatacaataccgcaaaaccgaggag  
gaacatcaacgctggcgcgaggcggtcgcgaaaaaccgcccgttgctacatccggtggtgcgaac  
gcatccggtgagcggtaaacaggcgctgtttgtgaatgaaggcttactacgcgaattggtgatg  
tgagcgagaaagagagcgaagccttgtaagttttgtttgcccataatcaccaaaccggagttt  
caggtgcgctggcgctggcaaccaaatagatattgcgatttgggataaccgctgaccagcacta  
tgccaatgccgattacctgccacagcgacggataatgcatcgggacgacgatccttggggataaac  
cgtttatcgggacggggtaa".replace('\n',")
```

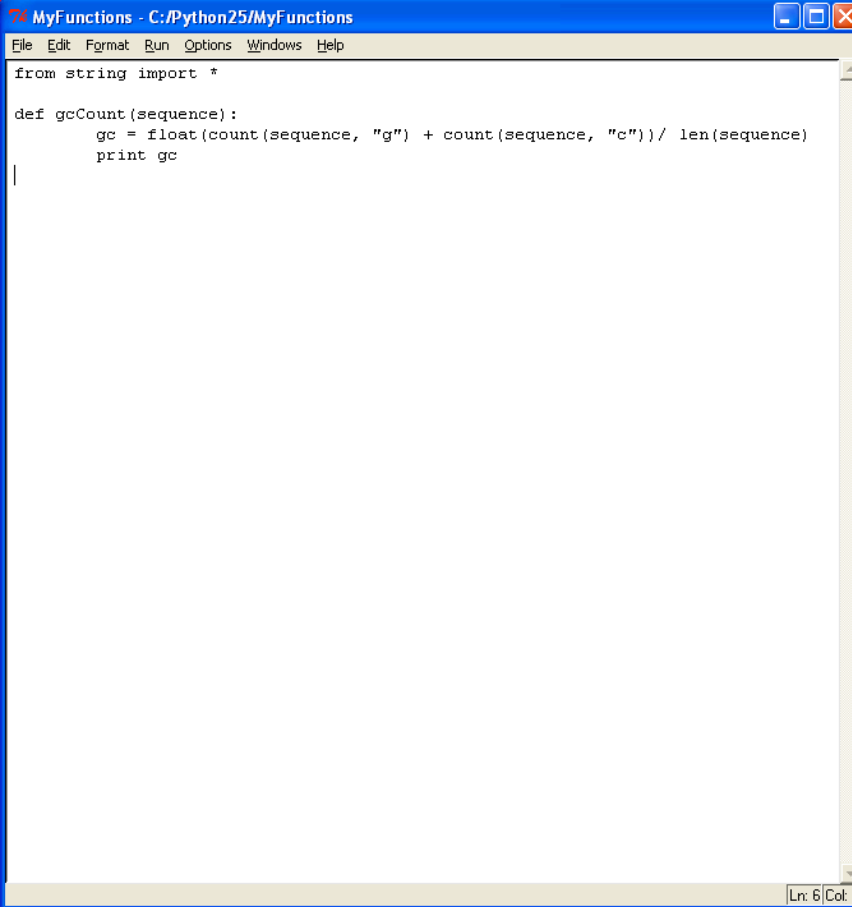
```
gc = float(count(cds, 'g') + count(cds, 'c'))/ len(cds)
```

```
print gc
```

Step 1: Wrap Reusable Code in Function

```
def gcCount(sequence):  
    gc = float(count(sequence, 'g') + count(sequence, 'c'))/ len(sequence)  
    print gc
```

Step 2: Add function to script file



```
MyFunctions - C:/Python25/MyFunctions
File Edit Format Run Options Windows Help
from string import *

def gcCount(sequence):
    gc = float(count(sequence, "g") + count(sequence, "c")) / len(sequence)
    print gc
|
Ln: 6 Col: 0
```

- *Save script in a file*
- *Re-load when you want to use the functions*
- *No need to retype your functions*
- *Keep a single group of related functions and declarations in each file*

Why Functions?

- Powerful mechanism for creating building blocks
- Code reuse
- Modularity
- Abstraction (i.e. hiding irrelevant detail)

Function Design Guidelines

- Should have a single well defined 'contract'
 - E.g. Return the gc-value of a sequence
- Contract should be easy to understand and remember
- Should be as general as possible
- Should be as efficient as possible
- Should not mix calculations with I/O

Applying the Guidelines

```
def gcCount(sequence):  
    gc = float(count(sequence, 'g') + count(sequence, 'c'))/ len(sequence)  
    print gc
```

What can be improved?

```
def gcCount(sequence):  
    gc = float(count(sequence, 'g') + count(sequence, 'c'))/ len(sequence)  
    return gc
```

Why is this better?

- More reusable function
- Can call it to get the *gcCount* and then decide what to do with the value
- May not have to *print* the value
- Function has ONE well-defined objective or CONTRACT

Decisional statements

*Indentation has meaning
in Python*

```
if <be1> :  
    <block1>  
elif <be2>:  
    <block2>  
...  
...  
else:  
    <blockn+1>
```

- *Each <be_i> is a BOOLEAN expressions*
- *Each <block_i> is a sequence of statements*
- *Level of indentation determines what's inside each block*

Compute the complement of a DNA base

```
def complementBase(base):  
    if (base == 'A'):  
        return 'T'  
    elif (base == 'T'):  
        return 'A'  
    elif (base == 'C'):  
        return 'G'  
    elif (base == 'G'):  
        return 'C'  
    else:  
        return 'X'
```

How can we improve this function?

Boolean Expressions

- Expressions that yield True or False values
- Ways to yield a Boolean value
 - Boolean constants: *True* and *False*
 - Comparison operators (>, <, ==, >=, <=)
 - Logical Operators (and, or, not)
 - Boolean functions
 - 0 (means False)
 - Empty string "" (means False)

A strange Boolean function

```
def test(x):  
    if x:  
        return True  
    else:  
        return False
```

What can you use this function for?

What types of values can it accept?

Some Useful Boolean Laws

- Lets assume that b, a are Boolean values:
 - $(b \text{ and True}) = b$
 - $(b \text{ or True}) = \text{True}$
 - $(b \text{ and False}) = \text{False}$
 - $(b \text{ or False}) = b$
 - $\text{not } (a \text{ and } b) = (\text{not } a) \text{ or } (\text{not } b)$
 - $\text{not } (a \text{ or } b) = (\text{not } a) \text{ and } (\text{not } b)$
- } De Morgan's Laws

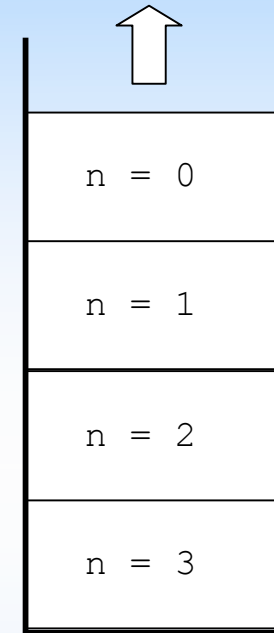
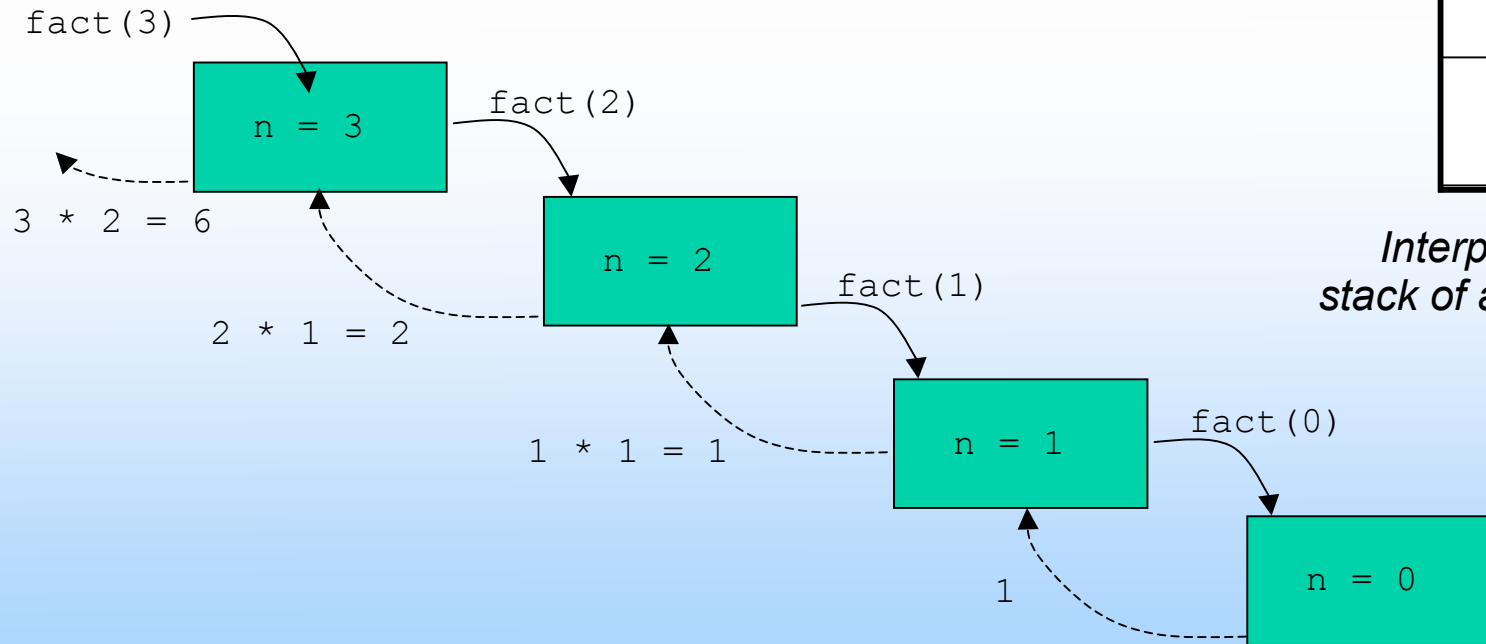
Recursive Functions

A classic!

```
>>> def fact(n):
...     if (n==0):
...         return 1
...     else:
...         return n * fact(n - 1)
...
>>> fact(5)
120
>>> fact(10)
3628800
>>> fact(100)
9332621544394415268169923885626670049071596826438162146859296389521
7599993229915608941463976156518286253697920827223758251185210916864
0000000000000000000000000000L
>>>
```


Recursion Basics

```
def fact(n):  
    if (n==0):  
        return 1  
    else:  
        return n * fact(n - 1)
```



Interpreter keeps a stack of activation records

Infinite Recursion

```
def fact(n):  
    if (n==0):  
        return 1  
    else:  
        return n * fact(n - 1)
```

What if you call fact 5.5? Explain

When using recursion always think about how will it stop or converge

Exercises on Functions

Write recursive Python functions to satisfy the following specifications:

1. Compute the number of x bases in a sequence where x is one of { C, T, G, A }
2. Compute the molecular mass of a sequence
3. Compute the complement of a sequence
4. Determine if two sequences are complement of each other
5. Compute the number of stop codons in a sequence
6. Determine if a sequence has a subsequence of length greater than n surrounded by stop codons
7. Return the starting position of the subsequence identified in exercise 6

Runtime Complexity

'Big O' Notation

```
def fact(n):  
    if (n==0):  
        return 1  
    else:  
        return n * fact(n - 1)
```

- *How 'fast' is this function?*
- *Can we come up with a more efficient version?*
- *How can we measure 'efficiency'?*
- *Can we compare algorithms independently from a specific implementation, software or hardware?*

Runtime Complexity

'Big O' Notation

Big Idea

Measure the number of steps taken by the algorithm as a asymptotic function of the size of its input

- What is a step?
- How can we measure the size of an input?
- Answer in both cases: YOU CAN DEFINE THESE!

'Big O' Notation

Factorial Example

- A 'step' is a function call to fact
- The size of an input value n is n itself

Step 1: Count the number of steps for input n

$$T(n) = T(n-1) + 1 = (T(n-2) + 1) + 1 = \dots = T(n-n) + n = T(0) + n = 0 + n = n$$

$T(0) = 0$

Step 2: Find the asymptotic function

$$T(n) = O(n)$$

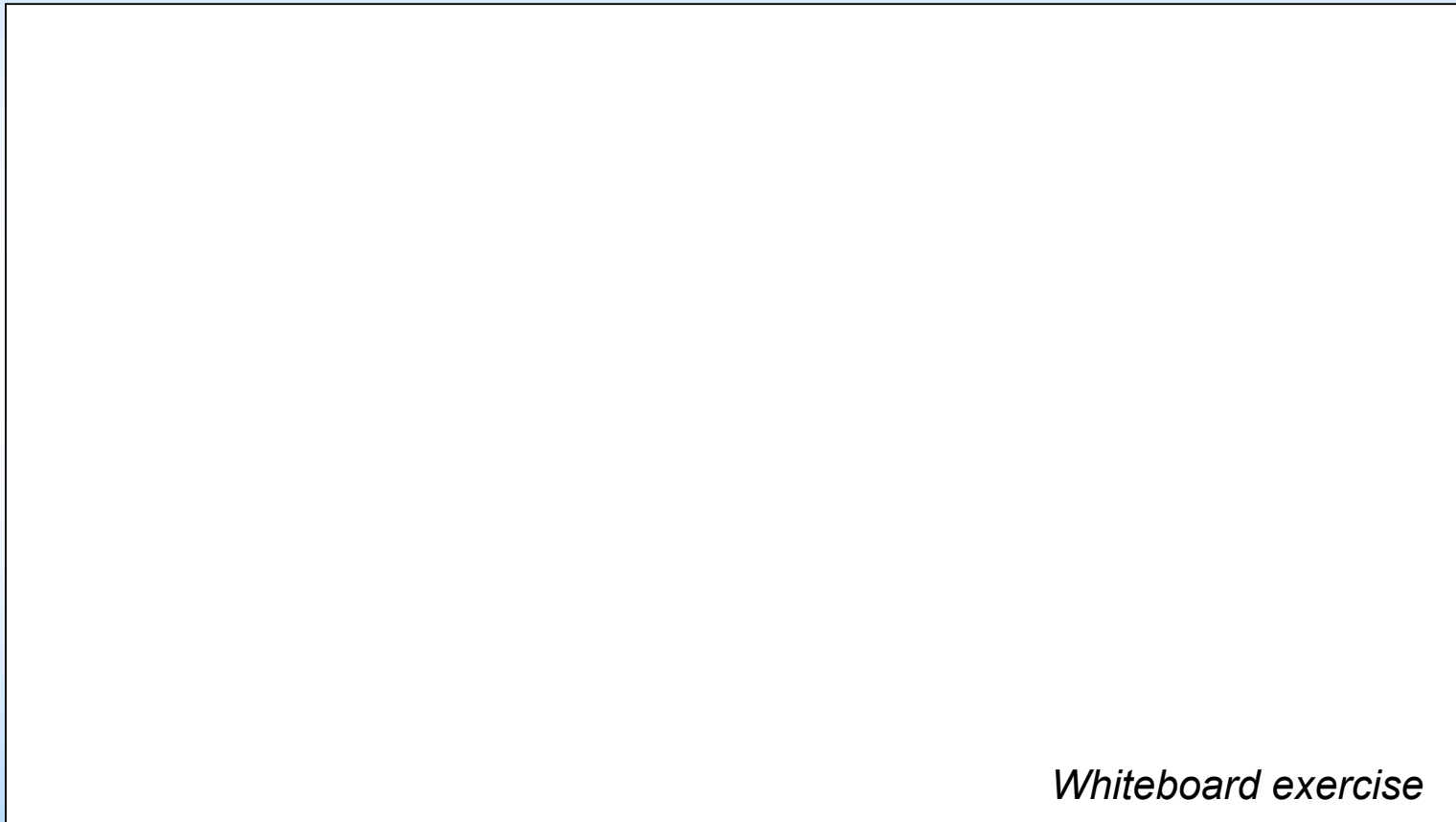
Another Classic

```
fibonacci(0) = 1  
fibonacci(1) = 1  
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2);
```

```
# Python version of Fibonacci  
def fibonacci (n):  
    if n == 0 or n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

Big O for Fibonacci

What is the runtime complexity of Fibonacci? Is this efficient?



Whiteboard exercise

Efficient ~ Polynomial Time complexity

Iteration

SYNTAX

while *<be>*:
<block>

SEMANTICS

*Repeat the execution of
<block> as long as
expression <be>
remains true*

SYNTAX = FORMAT
SEMANTICS = MEANING

Iterative Factorial

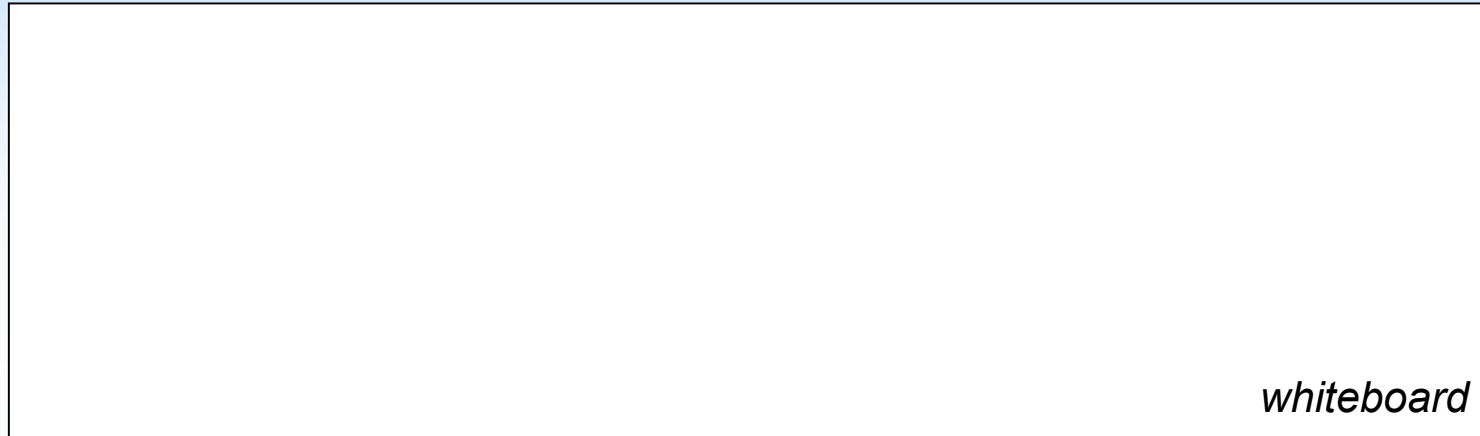
```
def iterFact(n):  
    result = 1  
    while(n>0):  
        result = result * n  
        n = n - 1  
    return result
```

Work out the runtime complexity:

whiteboard

Iterative Fibonacci

Code:

A large white rectangular area with a black border, intended for writing code. The word "whiteboard" is written in the bottom right corner.

whiteboard

Runtime complexity:

A large white rectangular area with a black border, intended for writing the runtime complexity. The word "whiteboard" is written in the bottom right corner.

whiteboard

Exercises on Functions

Write *iterative* Python functions to satisfy the following specifications:

1. Compute the number of x bases in a sequence where x is one of { C, T, G, A }
2. Compute the molecular mass of a sequence
3. Compute the complement of a sequence
4. Determine if two sequences are complement of each other
5. Compute the number of stop codons in a sequence
6. Determine if a sequence has a subsequence of length greater than n surrounded by stop codons
7. Return the starting position of the subsequence identified in exercise 6

Formatted Output using % operator

<format> % <values>

```
>>> '%s is %d years old' % ('John', 12)
'John is 12 years old'
>>>
```

- <format> is a string
- <values> is a list of values n parenthesis (a.k.a. a tuple)
- % produces a string replacing each %x with a correding value from the tuple

For more details visit: <http://docs.python.org/lib/typesseq-strings.html>

Bioinformatics Example

Description of the function's contract

```
def restrict(dna, enz):  
    'print all start positions of a restriction site'  
    site = find (dna, enz)  
    while site != -1:  
        print 'restriction site %s at position %d' % (enz, site)  
        site = find (dna, enz, site + 1)
```

```
>>> restrict(cds,'gccg')  
restriction site gccg at position 32  
restriction site gccg at position 60  
restriction site gccg at position 158  
restriction site gccg at position 225  
restriction site gccg at position 545  
restriction site gccg at position 774  
>>>
```

Is this a good name for this function?

The For Loop

Another Iteration Statement

SYNTAX

*for <var> in <sequence>:
<block>*

SEMANTICS

*Repeat the execution of
the <block> binding
variable <var> to each
element of the sequence*

For Loop Example

```
def iterFact2(n):  
    result = 1  
    for i in xrange(1,n+1):  
        result = result * i  
    return result
```

Xrange(start,end,step) generates a sequence of values :

- start = first value
- end = value right after last one
- step = increment

Nested For Loops

Example: Multiplication Table

```
def simpleMultiplicationTable(n):
```

```
    for i in xrange(0,n+1,1):
```



```
        for j in xrange(0,n+1):
```

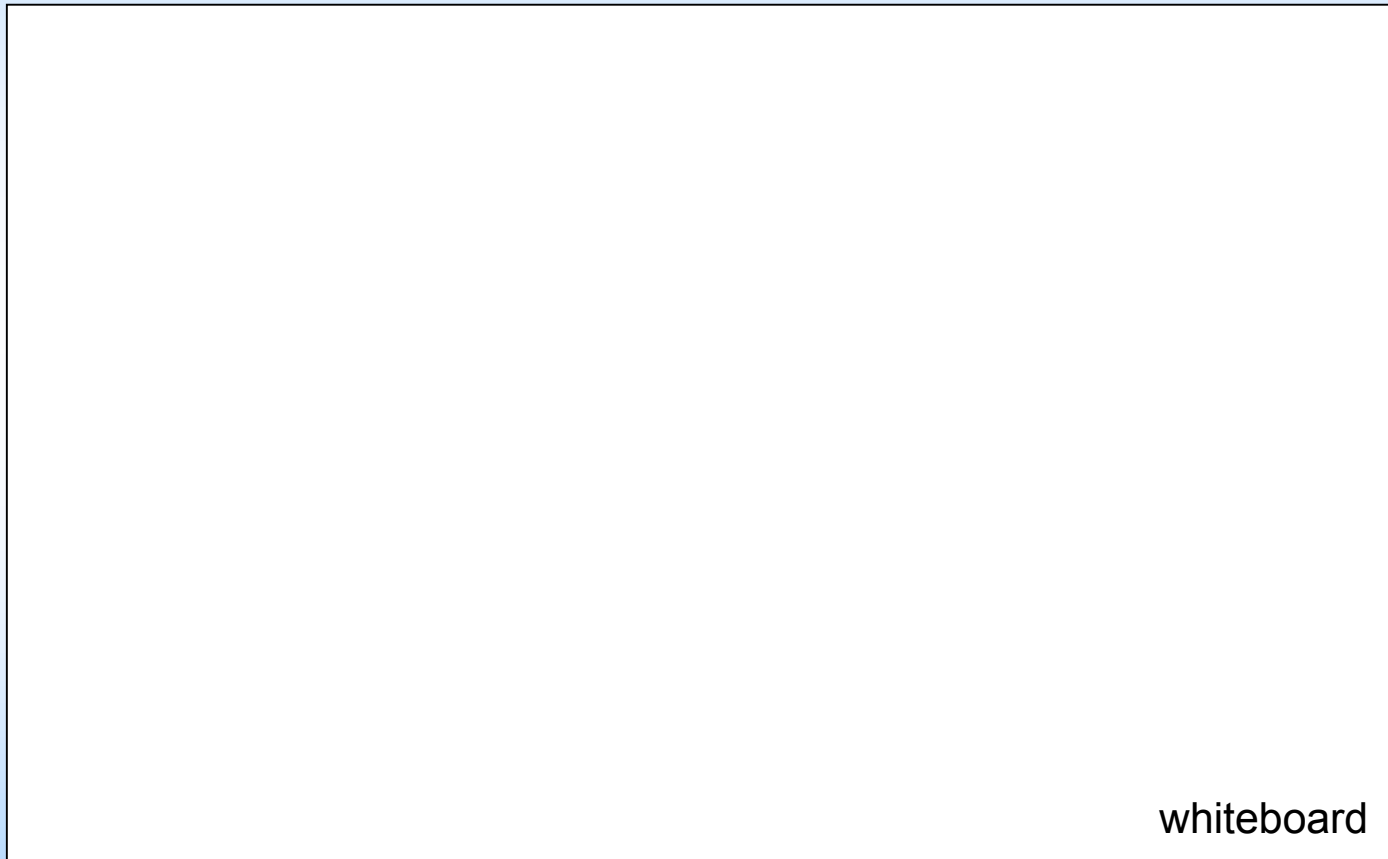


```
            print i, '*',j, '=', i*j
```

Semantics

*Inner loops iterates from beginning to end
for each single iteration of outer loop*

Improving the Format of the Table



whiteboard