Essential Computing for Bioinformatics

Lecture 4

High-level Programming with Python

Controlling the flow of your program

MARC: Developing Bioinformatics Programs
July 2009

Alex Ropelewski
PSC-NRBSC
Bienvenido Vélez
UPR Mayaguez

Reference: How to Think Like a Computer Scientist: Learning with Python

Essential Computing for Bioinformatics

- The following material is the result of a curriculum development effort to provide a set of courses to support bioinformatics efforts involving students from the biological sciences, computer science, and mathematics departments. They have been developed as a part of the NIH funded project “Assisting Bioinformatics Efforts at Minority Schools” (2T36 GM008789). The people involved with the curriculum development effort include:

- Dr. Hugh B. Nicholas, Dr. Troy Wymore, Mr. Alexander Ropelewski and Dr. David Deerfield II, National Resource for Biomedical Supercomputing, Pittsburgh Supercomputing Center, Carnegie Mellon University.
- Dr. Ricardo González Méndez, University of Puerto Rico Medical Sciences Campus.
- Dr. Alade Tokuta, North Carolina Central University.
- Dr. Jaime Seguel and Dr. Bienvenido Vélez, University of Puerto Rico at Mayagüez.
- Dr. Satish Bhalla, Johnson C. Smith University.

- Most recent versions of these presentations can be found at http://marc.psc.edu/

Outline

- Basics of Functions
- Decision statements
- Recursion
- Iteration statements

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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?

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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)
0.7853981633974483
>>> radians(180)
3.141592653589793
```

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Monolithic Code

```
From string import *

cds = “atgagtgaacgtctgagcattaccccgctggggccgtatatc”

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

print gc
```

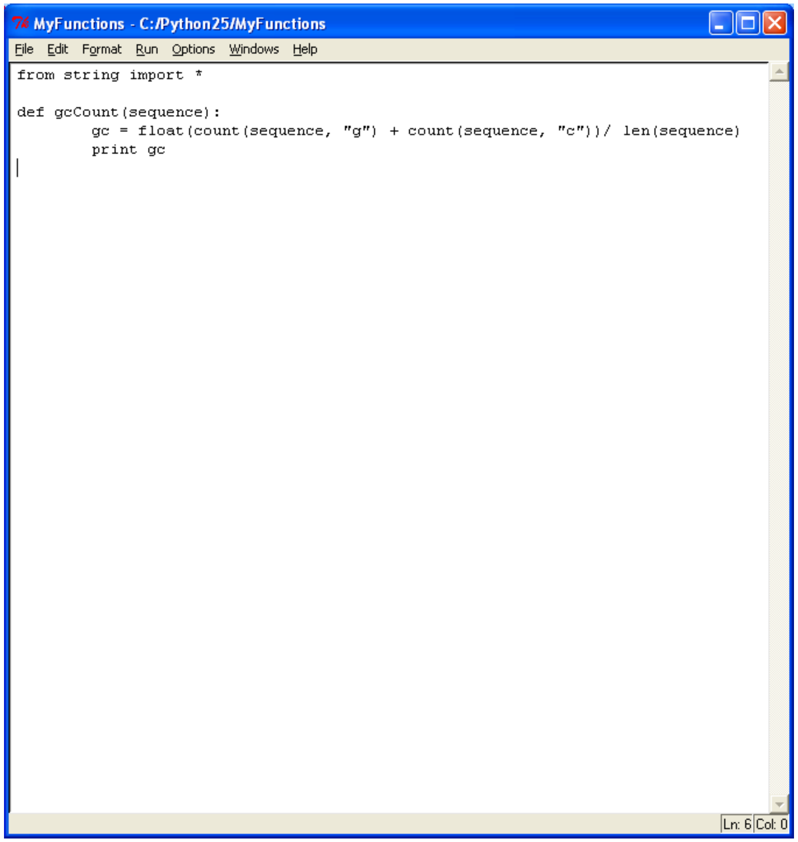
These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

```
>>> gcCount(“actgaccgggat”)
```

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Step 2: Add function to script file



```
from string import *

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

- *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*

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Outline

- ✓ Basics of Functions
- Decision statements
- Recursion
- Iteration statements

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Decision statements

Indentation has meaning in Python

```
if <$be_1$> :
    <$block_1$>
elif <$be_2$>:
    <$block_2$>
…
…
else:
    <$block_{n+1}$>
```

- *Each <$be_{i>}$ is a BOOLEAN expressions*
- *Each <$block_i$>is a sequence of statements*
- *Level of indentation determines what's inside each block*

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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'
```

How can we improve this function?

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Boolean Expressions

- Expressions that yield True of 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)

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Some Useful Boolean Laws

- Lets assume that b,a are Boolean values:
  - (b and True) = b
  - (b or True) = True
  - (b and False) = False
  - (b or False) = b
  - not (a and b) = (not a) or (not b)
  - not (a or b) = (not a) and (not b)

} De Morgan's Laws

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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?

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Outline

- ✓Basics of Functions
- ✓Decision statements
- Recursion
- Iteration statements

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Recursive Functions

A classic!

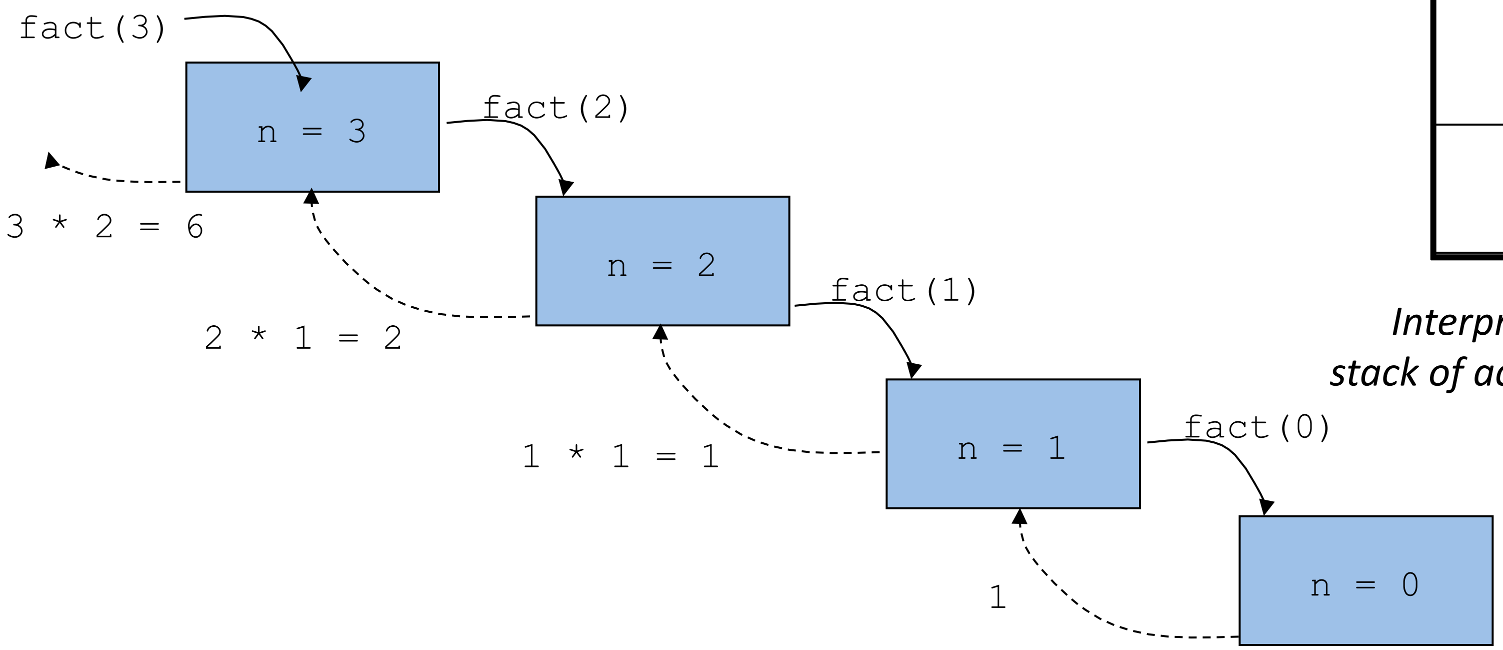
```
def fact(n):
    if (n==0):
        return 1
    else:
        return n * fact(n - 1)
```

```
>>> fact(5)
120
>>> fact(10)
3628800
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217599993
22991560894146397615651828625369792082722375825118521091686400000000000000
0000000000L
>>>
```

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Recursion Basics

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



fact(3)

n = 3

fact(2)

3 * 2 = 6

n = 2

fact(1)

2 * 1 = 2

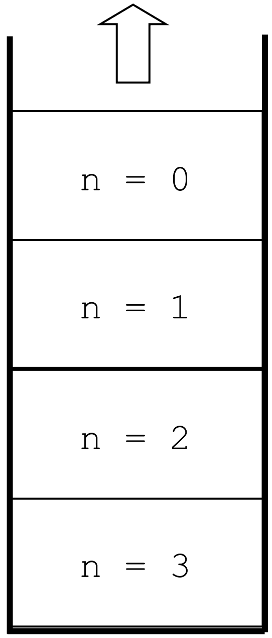
n = 1

fact(0)

1 * 1 = 1

n = 0

1



n = 0

n = 1

n = 2

n = 3

Interpreter keeps a
stack of activation records

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Beware of Infinite Recursions!

```
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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Write recursive Python functions to satisfy the following specifications:

- Compute the reverse of a sequence
- Compute the molecular mass of a sequence
- Compute the reverse complement of a sequence
- Determine if two sequences are complement of each other
- Compute the number of stop codons in a sequence
- Determine if a sequence has a subsequence of length greater than n surrounded by stop codons
- Return the starting position of the subsequence identified in exercise 6

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Reversing a sequence recursively

```
def reverse(sequence):

    'Returns the reverse string of the argument sequence'

    if (len(sequence)>1):

        return reverse(sequence[1:])+sequence[0]

    else:

        return sequence
```

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?*

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Big Idea

Measure the number of *steps* taken by the algorithm as an 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!

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

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

Step 1: Count the number of steps for input n

$$T(0) = 0$$

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

Step 2: Find the asymptotic function

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

A.K.A Linear Function

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

- ✓ Basics of Functions
- ✓ Decision statements
- ✓ Recursion
- Iteration statements

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Iteration

SYNTAX → *while <be>:*
<block>

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

SYNTAX = FORMAT
SEMANTICS = MEANING

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Iterative Factorial

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

Work out the runtime complexity:

whiteboard

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

```
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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Revisiting code from Lecture 1

```
seq="ACTGTCGTAT"
print seq
Acount= seq.count('A')
Ccount= seq.count('C')
Gcount= seq.count('G')
Tcount= seq.count('T')
Total = float(len(seq))
APct = int((Acount/Total) * 100)
print 'A percent = %d ' % APct
CPct = int((Ccount/Total) * 100)
print 'C percent = %d ' % CPct
GPct = int((Gcount/Total) * 100)
print 'G percent = %d ' % GPct
TPct = int((Tcount/Total) * 100)
print 'T percent = %d ' % TPct
```

Can we reduce the amount of repetitive code?

Approach: Use For Loop

```
bases = ['A', 'C', 'T', 'G']
sequence = "ACTGTCGTAT"
for base in bases:
    nextPercent = 100 * sequence.count(base)/float(len(sequence))
    print 'Percent %s: %d' % (base, nextPercent)
```

How many functions would you refactor this code into?

Write iterative Python functions to satisfy the following specifications:

1. Compute the reverse of a sequence
2. Compute the molecular mass of a sequence
3. Compute the reverse 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

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Finding Patterns Within Sequences

```
from string import *
def searchPattern(dna, pattern):
        'print all start positions of a pattern string inside a target string’
        site = find (dna, pattern)
        while site != -1:
                print ’pattern %s found at position %d' % (pattern, site)
                site = find (dna, pattern, site + 1)
```

```
>>> searchPattern("acgctaggct","gc")

pattern gc at position 2

pattern gc at position 7

>>>
```

Example from: *Pasteur Institute Bioinformatics Using Python*

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

- Extend searchPattern to handle unknown residues