Programming Applications

What is Computer Programming?

- An algorithm is a series of steps for solving a problem

- A programming language is a way to express our algorithm to a computer

- Programming is the process of writing instructions (i.e., an algorithm) to a computer for the purpose of solving a problem

We will be using the programming language Python

Variables and Types

- Variables store values of some type. Types have operators associated with them.

```
year = 2006
nextYear = year + 1
GC_content = 2.0 * 0.21
kozak = “ACC” + “ACCATGG”
year = year + 1
kozak = kozak + “TT” + kozak
```

- You can have the computer tell you the value of a variable

```
print nextYear
print “The GC content is: ”, GC_content
print year
print kozak
```

Strings

- Strings are a sequence of characters

```
protein = “MAFGHIWLVML”
```

- Strings are index-able

`protein[0]` refers to 'M', the first character in `protein`
`protein[4]` refers to 'H', the fifth character in `protein`

- Strings have lots of operations

`protein.lower()` returns "mafghiwlvml"
`protein.count('L')` returns 2
`protein.replace('L', '?')` returns "MAFGHIW?VM?"
`len(protein)` returns 11

Amino Acid Content

```
protein = “MAFGHIWLVML”
```

- What percent of the sequence corresponds to leucines?

```
numberOfLeucines = float(protein.count(‘L’))
totalAAs = float(len(protein))
freq_L = numberOfLeucines / totalAAs
print freq_L
```

Slicing a String

```
protein = “MAFGHIWLVML”

# Grab a piece of the sequence
firstThreeAAs = protein[0:3]
print firstThreeAAs
middleThreeAAs = protein[4:7]
print middleThreeAAs
finalThreeAAs = protein[len(protein)-3:len(protein)]
print finalThreeAAs
```

Booleans

```
protein = "MAFGHIWLVML"
```

- Booleans are either `True` or `False`

`protein == "MAFGHIWLVML"`	True
`protein != "MAFGHIWLVML"`	False
`protein != "MWPPWML"`	True
`protein == "mafghiwlvml"`	False
`protein.lower() == "mafghiwlvml"`	True
`len(protein) >= 11`	True
`len(protein) > 11`	False
`len(protein) <= 11`	False
`'I' in protein`	True
`'Z' in protein`	False
`'I' not in protein`	False
`'Z' not in protein`	True

Boolean Operators: and vs. or

Suppose A and B are boolean values.

Then "`A and B`" is true if both A is true *and* B is true.

Then "`A or B`" is true if either A is true *or* B is true.

```
x = True
y = False
z = True
```

`x and y`	False
`x or y`	True
`x and z`	True
`x or z`	True
`y or y`	False

Boolean Operator Examples

```
a = 17.1
b = -14.375
codon = “ATG”
```

`(a > 0) and (a < 20)`	True
`(b >= 0) or (b <= 20)`	True
`(b < a) and (a >= 0) and (b >= 0)`	False
`(a < 0) or (b > 0) or (a == b)`	False
`((a > 0) and (b > 0)) or ((codon == “ATG”) and (b < 0))`	True
`((codon == “ATG”) or (b < 0)) and ((a < 0) or (a == b))`	False

Reading in a Sequence File

```
file = open(“genome.txt”)
sequence = “”

# Read in first line of file and check for FASTA format
headerLine = file.readline()
if (headerLine[0] != ‘>’):    # First character is not ‘>’
        sequence = headerLine  # First line is part of seq.

# Read in the rest of the file (i.e., the sequence)
sequence = sequence + file.read()

# Remove all carriage returns from the sequence
sequence = sequence.replace(“\n”, “”)
```

Iteration Refresher

- Examples

```
# Assuming we have a coding sequence, print out each codon
startOfCodon = 0
while (startOfCodon < len(sequence)):
        codon = sequence[startOfCodon:startOfCodon+3]
        print codon
        startOfCodon = startOfCodon + 3
```

```
# Find the start of all possible ORFs in sequence
startOfCodon = 0
while (startOfCodon < len(sequence)):
        codon = sequence[startOfCodon:startOfCodon+3]
        if (codon == "ATG"):
                print "Found start codon at ", startOfCodon
        startOfCodon = startOfCodon + 1
```

Complementing a Sequence

```
# Complement the DNA string in the variable *sequence*
index = 0
comp = ""
while (index < len(sequence)):
        if (sequence[index] == 'A'):
                comp = comp + 'T'
        if (sequence[index] == 'C'):
                comp = comp + 'G'
        if (sequence[index] == 'G'):
                comp = comp + 'C'
        if (sequence[index] == 'T'):
                comp = comp + 'A'
        index = index + 1
print comp
```

- What if we want to complement lots of different sequences all through our program?

Defining Functions

- We can give a set of code (i.e., a set of instructions) a name "XYZ", and we can tell the computer to execute the "XYZ" instructions whenever we need those instructions

```
# Create a complemented version of the DNA string *sequence*
def complement(sequence):
        index = 0
        comp = ""
        while (index < len(sequence)):
                if (sequence[index] == 'A'):
                        comp = comp + 'T'
                if (sequence[index] == 'C'):
                        comp = comp + 'G'
                if (sequence[index] == 'G'):
                        comp = comp + 'C'
                if (sequence[index] == 'T'):
                        comp = comp + 'A'
                index = index + 1
        return comp
```

Using Functions

- We can now use our function whenever we like to complement a sequence

```
# Complement sequences with reckless abandon
s1 = "GGA"
complementedSequence = complement(s1)
print complementedSequence                          CCT

s2 = "TGTG"
s2_complemented = complement(s2)
print s2_complemented                               ACAC

s3 = "ATGCATGCGA"
print complement(s3)                                TACGTACGCT

s4 = "CCGATGC"
s4_complement = complement(s4)
print complement(s4_complement)                     CCGATGC
```

All Together

```
# Create a complemented version of the DNA string *sequence*
def complement(sequence):
            index = 0
            comp = ""
            while (index < len(sequence)):
                        if (sequence[index] == 'A'):
                                    comp = comp + 'T'
                        if (sequence[index] == 'C'):
                                    comp = comp + 'G'
                        if (sequence[index] == 'G'):
                                    comp = comp + 'C'
                        if (sequence[index] == 'T'):
                                    comp = comp + 'A'
                        index = index + 1
            return comp

# Complement sequences with reckless abandon
s1 = "GGA"
complementedSequence = complement(s1)
print complementedSequence

s2 = "TGTG"
s2_complemented = complement(s2)
print s2_complemented

s3 = "ATGCATGCGA"
print complement(s3)

s4 = "CCGATGC"
s4_complement = complement(s4)
print complement(s4_complement)
```

Reversing a Sequence

```
# Create a reversed version of the DNA string *sequence*
def reverse(sequence):
        index = 0
        rev = ""
        while (index < len(sequence)):
                rev = sequence[index] + rev
                index = index + 1
        return rev

# Reverse sequences
s1 = "GGA"
reversedSequence = reverse(s1)
print reversedSequence                                  AGG

s2 = "TGTG"
print reverse(s2)                                       GTGT
```

Reverse Complementing a Sequence

```
# Create a reverse complemented version of *sequence*
def reverseComplement(sequence):
        return reverse(complement(sequence))

# Reverse complement sequences
s1 = "GGA"
print reverseComplement(s1)                    TCC

s2 = "TGTG"
print reverseComplement(s2)                    CACA
```

All Together

```
# Create a complemented version of the DNA string *sequence*
def comp(sequence):
            index = 0
            comp = ""
            while (index < len(sequence)):
                        if (sequence[index] == 'A'):
                                    comp = comp + 'T'
                        if (sequence[index] == 'C'):
                                    comp = comp + 'G'
                        if (sequence[index] == 'G'):
                                    comp = comp + 'C'
                        if (sequence[index] == 'T'):
                                    comp = comp + 'A'
                        index = index + 1
            return comp

# Create a reversed version of the DNA string *sequence*
def reverse(sequence):
            index = 0
            rev = ""
            while (index < len(sequence)):
                        rev = sequence[index] + rev
                        index = index + 1
            return rev

# Create a reverse complemented version of *sequence*
def reverseComplement(sequence):
            return reverse(complement(sequence))

# Reverse complement sequences
s1 = "GGA"
print reverseComplement(s1)

s2 = "TGTG"
print reverseComplement(s2)
```

Functions

- Code can quickly become long and complicated
- Functions help code readability and generality
- When defining a function, the function may have parameters
- When calling a function, we must use an argument for each of the function's parameters
- Variables in functions are local to the function
- Functions can also return values

Example Function: Minimum

```
# Return the minimum of two numbers, a and b
def minimum(a, b):
        min = a
        if (b < a):
                min = b
        return min


# Examples using the "minimum" function
print minimum(5, -15)                                   -15

x = 7
y = 10
print minimum(x, y)                                     7
print minimum(y, x)                                     7

y = 5
print minimum(x, y)                                     5
print minimum(y, minimum(2, 12))                        2
```

Example Function: Random Sequences

```
import random

# Generate a random sequence of 20 DNA nucleotides.
# Each character in generated sequence has an equal
# chance (i.e., 25%) of being an adenine, cytosine,
# guanine, or thymine.
def generateRandomSequence():
   sequence = ""
   count = 0
   while (count < 20):
      random_number = random.random()
      if ((random_number >= 0.00) and (random_number < 0.25)):
         sequence = sequence + "A"
      if ((random_number >= 0.25) and (random_number < 0.50)):
         sequence = sequence + "C"
      if ((random_number >= 0.50) and (random_number < 0.75)):
         sequence = sequence + "G"
      if ((random_number >= 0.75) and (random_number < 1.00)):
         sequence = sequence + "T"

      count = count + 1
   return sequence
```

Example Function: Random Sequences

```
# Examples using the "generateRandomSequence" function
s = generateRandomSequence()
print s                                  AGAGCCGTACGAGTTCGATC

print generateRandomSequence()           TTACTTAGCGTAGGATCTCA

print generateRandomSequence()           CGTAGCTAGTCCATCGCGTA

s = generateRandomSequence()
print s                                  GTACGTCGTGTACGTCATCG
```

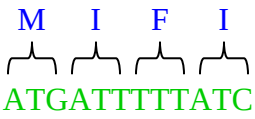
Translating a Codon

```
# Translate a codon into its amino acid
def translateCodon(codon):
        aa = '?'

        if (codon == "ATT"):
                aa = 'I'
        if (codon == "ATC"):
                aa = 'I'
        if (codon == "ATA"):
                aa = 'I'
        if (codon == "ATG"):
                aa = 'M'
        if (codon == "TTT"):
                aa = 'F'
        if (codon == "TTC"):
                aa = 'F'

        ...

        return aa
```



Open Reading Frames (ORFs)

GACTTATGCTCATGCACTGTACCTAAGATGGCTGACA

ORF Hunting

```
# Find all ORFs in *sequence*
startIndex = 0
while (startIndex < len(sequence)):
        startCodon = sequence[startIndex:startIndex+3]
        if (startCodon == “ATG”):  # We found a start codon









        startIndex = startIndex + 1
```

ORF Hunting

```
# Find all ORFs in *sequence*
startIndex = 0
while (startIndex < len(sequence)):
        startCodon = sequence[startIndex:startIndex+3]
        if (startCodon == “ATG”):  # We found a start codon

                # Let’s search for a stop codon
                stopIndex = startIndex + 3
                while (stopIndex < len(sequence)):
                        stopCodon = sequence[stopIndex:stopIndex+3]
                        if ((stopCodon==“TAA”) or (stopCodon==“TAG”)
                                or (stopCodon==“TGA”)): # We have an ORF

                                # Print out the ORF
                                print sequence[startIndex:stopIndex+3], “\n”

                                # Terminate the current search for stop codons
                                stopIndex = len(sequence)

                        stopIndex = stopIndex + 3

        startIndex = startIndex + 1
```

Genomic Sequence Motifs

- TATA boxes and Kozak sequences are examples of sequence *motifs*
- The *consensus* Kozak sequence is ACCATGG
- While the fourth, fifth, and sixth nucleotides in the Kozak sequence are almost always ATG, the first nucleotide is usually an adenine but sometimes a guanine. Sometimes, the Kozak sequence is expressed as RCCATGG where R stands for any purine nucleotide.
- Another means for representing a motif is with a *weight-matrix*, which indicates the frequency of each nucleotide in each position of the motif.

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

Probability of Motif Instances

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

GGGTCCC

ACCATGG

GCCATGG

TTTTTTT

Probability of Motif Instances

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

Probability(GGGTCCC) = 0.43 * 0.23 * 0.25 * 0.01 * 0.01 * 0.01 * 0.20 = 4.9×10^{-9}

Probability(ACCATGG) =

Probability(GCCATGG) =

Probability(TTTTTTT) =

Probability of Motif Instances

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

Probability(GGGTCCC) =

Probability(ACCATGG) = 0.52 * 0.30 * 0.50 * 0.97 * 0.97 * 0.97 * 0.40 = 2.8×10^{-2}

Probability(GCCATGG) =

Probability(TTTTTTT) =

Probability of Motif Instances

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

Probability(GGGTCCC) =

Probability(ACCATGG) =

Probability(GCCATGG) = 0.43 * 0.30 * 0.50 * 0.97 * 0.97 * 0.97 * 0.40 = 2.4×10^{-2}

Probability(TTTTTTT) =

Probability of Motif Instances

Example weight matrix for Kozak sequence

	1	2	3	4	5	6	7
A	52%	27%	13%	97%	1%	1%	30%
C	3%	30%	50%	1%	1%	1%	20%
G	43%	23%	25%	1%	1%	97%	40%
T	2%	20%	12%	1%	97%	1%	10%

Probability(GGGTCCC) =

Probability(ACCATGG) =

Probability(GCCATGG) =

Probability(TTTTTTT) = 0.02 * 0.20 * 0.12 * 0.01 * 0.97 * 0.01 * 0.10 = 4.7×10^{-9}