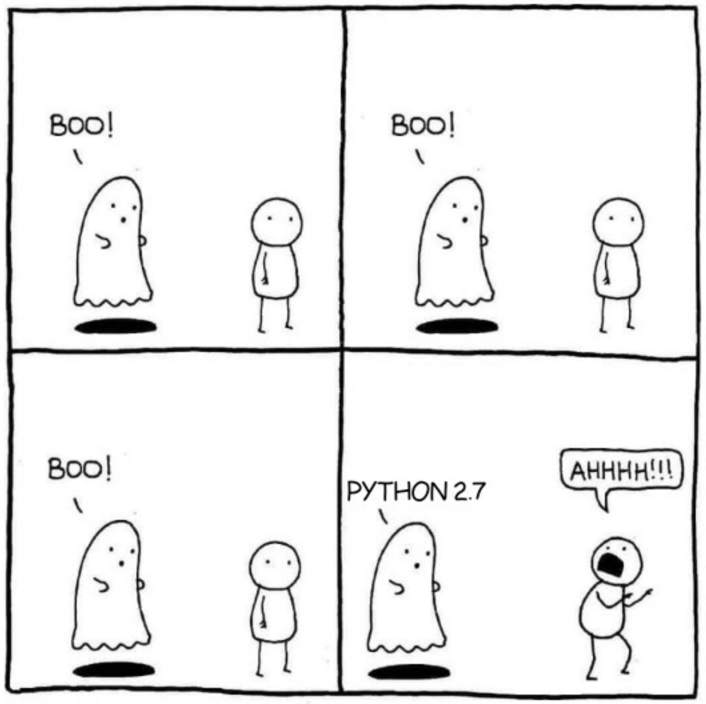
Lecture 12: An introduction to Python and its use in Bioinformatics

Edited by Roberto Villegas-Diaz

[Found online]

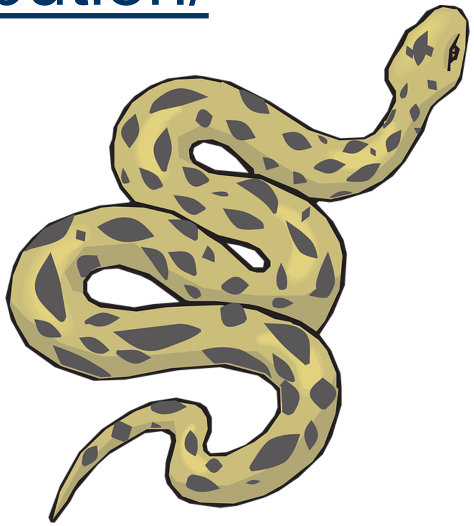
But first, some Python blah blah …



Repository: https://github.com/villegar/BIO792

Software

- Base Python (3.8.1): https://www.python.org
- Anaconda “The World's Most Popular Python/R Data Science Platform”: https://www.anaconda.com/distribution/
- Python Package Index (~218k) https://pypi.org



if Statement

- if *expression*:

        action

Example:

```
a1 = 'A'; a2 = 'C';
match = 0;
if (a1 == a2) :
   match+=1;
```

if-elif-else Statement

- if *expression*:

  ```
  if expression:
          action 1
  elif expression:
          action 2
  else :
          action 3
  ```

Example:

```
a1 = 'A'; a2 = 'C';
match = 0; gap = 0;
if (a1 == a2) :
    match+=1;
elif (a1 > a2):
    # Do something
else:
    gap+=1;
```

String operations

mystring = “Hello World!”		
Expression	Value	Purpose
`len(mystring)`	`12`	number of characters in mystring
`“hello”+“world”`	`“helloworld”`	Concatenate strings
`“%s world”%“hello”`	`“hello world”`	Format strings (like sprintf)

`“world” == “hello”`
`“world” == “world”`
	`0 or False`
	`1 or True`
		Test for equality

`“a” < “b”`
`“b” < “a”`
	`1 or True`
	`0 or False`
		Alphabetical ordering

Lists

mylist=[“a”,”b”,3.58,”d”,4,0]		
mylist[0] mylist[2]	a 3.58	Indexing
mylist[-1] mylist[-2]	0 4	Negative indexing (counts from end)
mylist[1:4]	[“b”,3.58,”d”]	Slicing (like strings)
“b” in mylist “e” not in mylist	1 or True 1 or True	
mylist.append(8)	[“a”,”b”,3.58,”d”,4,0,8]	Add to end of list

Dictionaries

mydict={“r”:1,”g”:2,”y”:3.5,8.5:8,9:”nine”}		
mydict.keys()	['y', 8.5, 'r', 'g', 9]	List of the keys
mydict.values()	[3.5, 8, 1, 2, 'nine']	List of the values
mydict[“y”]	3.5	Value lookup
mydict.has_key(“r”)	True or 1	Check for keys
mydict.update({“a”:75})	{8.5: 8, 'a': 75, 'r': 1, 'g': 2, 'y': 3.5, 9: 'nine'}	Add pairs to dictionary

for Statement

```
for var in list:
    action
```

- *Sets var to each item in list and performs action*
- *range() function generates lists of numbers:*
  range (5) -> [0,1,2,3,4]

Example

```
mylist=["hello","hi","hey","!"];
for i in mylist:
    print i
```

```
Iteration 1 prints:  hello
Iteration 2 prints:  hi
Iteration 3 prints:  hey
Iteration 4 prints:  !
```

while Statement

while expression:

    action

<u>Example</u>

x = 0;

while x != 3:

    x = x + ~~1~~ 2

Infinite loop!

Iteration 1: x=0+1=1

Iteration 2: x=1+1=2

Iteration 3: x=2+1=3

Iteration 4: don’t exec

Example: Amino Acid Search

- Write a program to count the number of occurrences of an amino acid in a sequence.
  - The program should prompt the user for
    - A sequence of amino acids (seq)
    - The search amino acid (aa)
  - The program should display the number of times the search amino acid (aa) occurred in the sequence (seq)

Example: Amino Acid Search (2)

```
#this program will calculate the number of occurrences of an amino acid
in a sequence


done=0
while (not done):
    sequence=input("Please enter a sequence: ");
    aa=input("Please enter the amino acid to look for: ");
```

Example: Amino Acid Search (3)

```
#compute the number of occurrences using for loop
cnt=0
for i in sequence:
    if i == aa:
        cnt+=1
if cnt == 1:
    print("%s occurs in that sequence once" % aa)
else:
    print("%s occurs in that sequence %d times" % (aa, cnt))
answer=input("try again? [yn] ")
if answer == "n" or answer == "N":
    done = 1
```

Python List Comprehensions

- Precise way to create a list
- Consists of an expression followed by a for clause, then zero or more for or if clauses
- Ex:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

- Ex:

```
>>> x = "acactgacct"
>>> y = [int(i=='c' or i=='g') for i in x]
>>> y
[0, 1, 0, 1, 0, 1, 0, 1, 1, 0]
>>> sum(y)/len(y)*100
50.0
```

Creating 2-D Lists

- To create a 2-D list L, with C columns and R rows initialized to 0:

```
L = [[]]           #empty 2-Dlist
L = [[0 for col in range(C)] for row in
  range(R)]
```

- To assign the value 5 to the element at the 2nd row and 3rd column of L

```
L[2][3] = 5
```

Zip – for parallel traversals

- Visit multiple sequences in parallel
- Ex:

```
>>> L1 = [1,2,3]
>>> L2 = [5,6,7]
>>> zip(L1, L2)
[(1,5), (2,6), (3,7)]
```

- Ex:

```
>>> for(x,y) in zip(L1, L2):
...        print(x, y, '--', x+y)
1 5 -- 6
2 6 -- 8
3 7 -- 10
```

Dictionary Construction with zip

- Ex:

```
>>> keys = ['a', 'b', 'd']
>>> vals = [1.8, 2.5, -3.5]
>>> hydro = dict(zip(keys,vals))
>>> hydro
{'a': 1.8, 'b': 2.5, 'd': -3.5}
```

File I/O

- To open a file
  - myfile = open('pathname', <mode>)
    - modes:

      'r' = read

      'w' = write
  - Ex: infile = open("D:\\Docs\\test.txt", 'r')
  - Ex: outfile = open("out.txt", 'w') – in same directory

Common input file operations

Operation	Interpretation
input = open ('file', 'r')	open input file
S = input.read()	read entire file into string S
S = input.read(N)	Read N bytes (N>= 1)
S = input.readline()	Read next line
L = input.readlines()	Read entire file into list of line strings

Common output file operations

Operation	Interpretation
output = open('file', 'w')	create output file
output.write(S)	Write string S into file
output.writelines(L)	Write all line strings in list L into file
output.close()	Manual close (good habit)

Processing *tblastn* output

```
countHits = 0
with open("my_tblastn_output_nr.txt","r") as tblastn:
    for hit in tblastn.readlines():
        hit = hit.split('\t')
        countHits += 1
        print("Sequence ID: %s" % hit[4])
        print("e-value: %s " % hit[7])
print("Number of hits: %s" % countHits)
tblastn.close()
```

```
Sequence ID:
NP_188918;NP_001326409;NP_001319620;Q9LUI2;BAB01254;AEE76677;ANM643
76;ANM64377
e-value: 100.000
```

Extracting data from string – split

- String.split([sep, [maxsplit]]) - Return a list of the words of the string *s*.
- If the optional argument *sep* is absent or None, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, formfeed).
- If the argument *sep* is present and not None, it specifies a string to be used as the word separator.
- The optional argument *maxsplit* defaults to 0. If it is nonzero, at most *maxsplit* number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most *maxsplit* +1 elements).

Split

- Ex:

```
>>> x = "a,b,c,d"
>>> x.split(',’)
['a', 'b', 'c', 'd']
>>> x.split(',’,2)
['a', 'b', 'c,d']
```

- Ex:

```
>>> y = "5 33    a     4"
>>> y.split()
['5', '33', 'a', '4']
```

Functions

- Function definition

```
def adder(a, b, c): return a+b+c
```

- Function calls

```
adder(1, 2, 3) -> 6
```

Functions – Polymorphism

```
>>>def fn2(c):
…               a = c * 3
…               return a
>>> print(fn2(5))
15
>>> print(fn2(1.5))
4.5
>>> print(fn2([1,2,3]))
[1,2,3,1,2,3,1,2,3]
>>> print(fn2("Hi"))
HiHiHi
```

Functions - Recursion

```
def fn_Rec(x):
   if x == []:
       return
   fn_Rec(x[1:])
   print(x[0])

y = [1,2,3,4]
fn_Rec(y)
```

```
4
3
2
1
```

Read FASTA file

```
SeqID = ""
A = 0
C = 0
G = 0
T = 0
```

Read FASTA file (2)

```
with open("exampledna.fasta","r") as fasta:
    for line in fasta.readlines():
        if(line.startswith(">")):
            SeqID = line.strip()
        else:
            line = line.upper()
            A += sum([int(i == 'A') for i in line])
            C += sum([int(i == 'C') for i in line])
            G += sum([int(i == 'G') for i in line])
            T += sum([int(i == 'T') for i in line])
GCcontent = (C + G)/(A + C + G + T)*100
```

Read FASTA file (3)

```
print("Sequence ID: %s" % SeqID)
print("A: %s \nC: %s \nG: %s \nT: %s" % (A,C,G,T))
print("GC content: %.2f" % GCcontent)
fasta.close()
```

```
Sequence ID: >AY117270.1 Arabidopsis thaliana unknown
protein (At4g13345) mRNA, complete cds
A: 318
C: 231
G: 284
T: 383
GC content: 42.35
```

Lecture 13: More fun

Lecture 13: Biopython Tutorial

Jeff Chang, Brad Chapman,
Iddo Friedberg, Thomas Hamelryck,
Michiel de Hoon, Peter Cock,
Tiago Antao, Eric Talevich,
Bartek Wilczyński

http://biopython.org/

What is Biopython?

- The Biopython Project is an international association of developers of freely available Python (https: //www.python.org) tools for computational molecular biology.
- Basically, the goal of Biopython is to make it as easy as possible to use Python for bioinformatics by creating high-quality, reusable modules and classes. Biopython features include parsers for various Bioinformatics file formats (BLAST, Clustalw, FASTA, Genbank,...), access to online services (NCBI, Expasy,...), interfaces to common and not-so-common programs (Clustalw, DSSP, MSMS...), a standard sequence class, various clustering modules, a KD tree data structure etc. and even documentation.

Supported formats

- Blast output – both from standalone and WWW Blast
- Clustalw
- FASTA
- GenBank
- PubMed and Medline
- ExPASy files, like Enzyme and Prosite
- SCOP, including ‘dom’ and ‘lin’ files
- UniGene
- SwissProt

What else?

- More details here: http://biopython.org/DIST/docs/tutorial/Tutorial.html#htoc3

Installing and TestingBiopython

```
pip install biopython
```

```
>>> import Bio
>>> print(Bio.__version__)
1.76
```

Working with sequences

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq(AGTACACTGGT’)
>>> print(my_seq)
AGTACACTGGT
>>> my_seq.alphabet
Alphabet()
```

Working with sequences (2)

```
>>> my_seq.complement()
Seq('TCATGTGACCA')
>>> my_seq.reverse_complement()
Seq('ACCAGTGTACT')
```

Simple FASTA parsing example

- Lady Slipper Orchids
- File: *ls_orchid.fasta*

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.fasta", "fasta"):
       print(seq_record.id)
       print(repr(seq_record.seq))
       print(len(seq_record))
```

```
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', SingleLetterAlphabet())
740
gi|2765657|emb|Z78532.1|CCZ78532
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAACAG...GGC', SingleLetterAlphabet())
753
```

Simple GenBank parsing example

- File: *ls_orchid.gbk*

```
from Bio import SeqIO
for seq_record in SeqIO.parse("ls_orchid.gbk", "genbank"):
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```

```
Z78533.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC', IUPACAmbiguousDNA())
740
Z78532.1
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAACAG...GGC', IUPACAmbiguousDNA())
753
```

More sequences: Alphabets

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("AGTACACTGGT", IUPAC.unambiguous_dna)
>>> my_seq
Seq('AGTACACTGGT', IUPACUnambiguousDNA())
>>> my_seq.alphabet
IUPACUnambiguousDNA()
```

More sequences: Alphabets (2)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_prot = Seq("AGTACACTGGT", IUPAC.protein)
>>> my_prot
Seq('AGTACACTGGT', IUPACProtein())
>>> my_prot.alphabet
IUPACProtein()
```

Sequences act like strings

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCG", IUPAC.unambiguous_dna)
>>> for index, letter in enumerate(my_seq):
... print("%i %s" % (index, letter))
0 G
1 A
2 T
3 C
4 G
>>> print(len(my_seq))
5
```

Sequences act like strings (2)

```
>>> from Bio.Seq import Seq
>>> "AAAA".count("AA")
2
>>> Seq("AAAA").count("AA")
2
```

Sequences act like strings (3)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> len(my_seq)
32
>>> my_seq.count("G")
9
>>> 100 * float(my_seq.count("G") + my_seq.count("C")) / len(my_seq)
46.875
```

Sequences act like strings (4)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqUtils import GC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> GC(my_seq)
46.875
```

Slicing a sequence

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
>>> my_seq[4:12]
Seq('GATGGGCC', IUPACUnambiguousDNA())

>>> my_seq[0::3]
Seq('GCTGTAGTAAG', IUPACUnambiguousDNA())
>>> my_seq[1::3]
Seq('AGGCATGCATC', IUPACUnambiguousDNA())
>>> my_seq[2::3]
Seq('TAGCTAAGAC', IUPACUnambiguousDNA())

>>> my_seq[::-1]
Seq('CGCTAAAAGCTAGGATATATCCGGGTAGCTAG', IUPACUnambiguousDNA())
```

Turning Seq objects into strings

```
>>> str(my_seq)
’GATCGATGGGCCTATATAGGATCGAAAATCGC’

>>> print(my_seq)
GATCGATGGGCCTATATAGGATCGAAAATCGC

>>> str(my_seq)
>>> print(my_seq)
>>> fasta_format_string = ">Name\n%s\n" % my_seq
>>> print(fasta_format_string)
>Name
GATCGATGGGCCTATATAGGATCGAAAATCGC
<BLANKLINE>
```

Concatenating or adding sequences

```
>>> from Bio.Alphabet import IUPAC
>>> from Bio.Seq import Seq
>>> protein_seq = Seq("EVRNAK", IUPAC.protein)
>>> dna_seq = Seq("ACGT", IUPAC.unambiguous_dna)
>>> protein_seq + dna_seq
Traceback (most recent call last):
...
TypeError: Incompatible alphabets IUPACProtein() and
IUPACUnambiguousDNA()
```

Concatenating or adding sequences (2)

```
>>> from Bio.Alphabet import generic_alphabet
>>> protein_seq.alphabet = generic_alphabet
>>> dna_seq.alphabet = generic_alphabet
>>> protein_seq + dna_seq
Seq('EVRNAKACGT')
```

Concatenating or adding sequences (3)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> list_of_seqs = [Seq("ACGT", generic_dna),
Seq("AACC", generic_dna), Seq("GGTT", generic_dna)]
>>> sum(list_of_seqs, Seq("", generic_dna))
Seq('ACGTAACCGGTT', DNAAlphabet())
```

Changing case

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> dna_seq = Seq("acgtACGT", generic_dna)
>>> dna_seq
Seq('acgtACGT', DNAAlphabet())
>>> dna_seq.upper()
Seq('ACGTACGT', DNAAlphabet())
>>> dna_seq.lower()
Seq('acgtacgt', DNAAlphabet())
```

Transcription

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG", 
IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> messenger_rna = coding_dna.transcribe()
>>> messenger_rna
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG', IUPACUnambiguousRNA())
```

Translation

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG",
IUPAC.unambiguous_dna)
>>> coding_dna
Seq('ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG', IUPACUnambiguousDNA())
>>> coding_dna.translate()
Seq('MAIVMGR*KGAR*', HasStopCodon(IUPACProtein(), '*'))
```

Translation (2)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> gene = 
Seq("GTGAAAAAGATGCAATCTATCGTACTCGCACTTTCCCTGGTTCTGGTCGCTCCCATGGCA" + \
... "GCACAGGCTGCGGAAATTACGTTAGTCCCGTCAGTAAAATTACAGATAGGCGATCGTGAT" + \
... "AATCGTGGCTATTACTGGGATGGAGGTCACTGGCGCGACCACGGCTGGTGGAAACAACAT" + \
... "TATGAATGGCGAGGCAATCGCTGGCACCTACACGGACCGCCGCCACCGCCGCGCCACCAT" + \
... "AAGAAAGCTCCTCATGATCATCACGGCGGTCATGGTCCAGGCAAACATCACCGCTAA",
... generic_dna)
>>> gene.translate(table="Bacterial")
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HR*',
HasStopCodon(ExtendedIUPACProtein(), '*'))

>>> gene.translate(table="Bacterial", to_stop=True)
Seq('VKKMQSIVLALSLVLVAPMAAQAAEITLVPSVKLQIGDRDNRGYYWDGGHWRDH...HHR',
ExtendedIUPACProtein())
```

Comparing sequences

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> seq1 = Seq("ACGT", IUPAC.unambiguous_dna)
>>> seq2 = Seq("ACGT", IUPAC.ambiguous_dna)
>>> str(seq1) == str(seq2)
True
>>> str(seq1) == str(seq1)
True
```

Comparing sequences (2)

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna, generic_protein
>>> dna_seq = Seq("ACGT", generic_dna)
>>> prot_seq = Seq("ACGT", generic_protein)
>>> dna_seq == prot_seq
BiopythonWarning: Incompatible alphabets DNAAlphabet() and
ProteinAlphabet()
True
```

MutableSeq objects

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA", IUPAC.unambiguous_dna)

>>> my_seq[5] = "G"
Traceback (most recent call last):
...
TypeError: ’Seq’ object does not support item assignment

>>> mutable_seq = my_seq.tomutable()
>>> mutable_seq
MutableSeq(’GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA’, IUPACUnambiguousDNA())

>>> new_seq = mutable_seq.toseq()
>>> new_seq
Seq(’AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG’, IUPACUnambiguousDNA())
```

MutableSeq objects (2)

```
>>> from Bio.Seq import MutableSeq
>>> from Bio.Alphabet import IUPAC
>>> mutable_seq = MutableSeq("GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA",
IUPAC.unambiguous_dna)

>>> mutable_seq
MutableSeq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq[5] = "C"
>>> mutable_seq
MutableSeq('GCCATCGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.remove("T")
>>> mutable_seq
MutableSeq('GCCACGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPACUnambiguousDNA())
>>> mutable_seq.reverse()
>>> mutable_seq
MutableSeq('AGCCCGTGGGAAAGTCGCCGGGTAATGCACCG', IUPACUnambiguousDNA())
```

UnknownSeq objects

- Only the length of the sequence is known

```
>>> from Bio.Seq import UnknownSeq
>>> unk = UnknownSeq(20)
>>> unk
UnknownSeq(20, character=’?’)
>>> print(unk)
????????????????????
>>> len(unk)
20
```

UnknownSeq objects (2)

- Define unknown character (i.e N or X)

```
>>> from Bio.Seq import UnknownSeq
>>> from Bio.Alphabet import IUPAC
>>> unk_dna = UnknownSeq(20, alphabet=IUPAC.ambiguous_dna)
>>> unk_dna
UnknownSeq(20, alphabet=IUPACAmbiguousDNA(), character=’N’)
>>> print(unk_dna)
NNNNNNNNNNNNNNNNNNNN
>>> unk_protein = unk_dna.translate()
>>> unk_protein
UnknownSeq(6, alphabet=ProteinAlphabet(), character=’X’)
```

Enough for today!

It's the end of the line, programmer...
...and I am missing the semicolon! Thanks!
TAP TAP
TAP
Daniel Stori {turnoff.us}