Let's start by importing from the **data** module:

In [1]:
```
from data import blast_out,human_sequence,IUPAC_codes
```

Problem 1:

The ***human_sequence*** variable imported from the **data** module is a cDNA sequence string resembling a typical aligned sequence, with gaps spread throughout.

a) Define a function capable of taking any sequence string as input, and that prints a new sequence without gaps. The function must be able to take the gap symbol as an argument (e.g., "-" or "?", etc)

In [45]:
```
# First method
def gap_cutter (sequence,gap_symbol):
    new_sequence = sequence.lower().replace(gap_symbol.lower(),"") # Using the .lower() or .upp
    print new_sequence                                             # the function is insensitive

gap_cutter(human_sequence,"-")

# Second method
def gap_cutter2 (sequence,gap_symbol):
    new_sequence = ""
    for character in sequence:
        if character.upper() != gap_symbol.upper():
            new_sequence += character
        else:
            pass
    print new_sequence

gap_cutter2(human_sequence,"-")
```

b) Define a function similar to the one created for a), but capable of printing a new sequence without gaps entirely OR without gaps only at the 3' and 5' extremities of the sequence.

Tips:

- You can use a boolean-like variable as an argument of the function, e.g. "zero_gaps = True" for when you want the function to remove gaps from both ends, and "zero_gaps = Right/Left" when, you want to remove them from only one side).

In [4]:
```
def gap_cutter3 (sequence,gap_symbol,zero_gaps):
    if zero_gaps == True:
        new_sequence = sequence.lower().replace(gap_symbol,"")
        print new_sequence
    elif zero_gaps == False:
        new_sequence = sequence.upper().strip(gap_symbol)
        print new_sequence

gap_cutter3 (human_sequence,"-",True)
gap_cutter3 (human_sequence,"-",False)
```

c) Define a function capable of finding any motif in a sequence, and returning a tuple with the starting and ending positions of its first appearance. If the motif does not exist, print an error message. Find the following motifs:

- "GGGTTCACTT"
- "GATCA"
- "AACAT"
- "GGTGTGGGGGG"

Tips:

- The function can be defined with two arguments: One for the sequence, and another for the motif

In [38]:
```
def motif_finder (motif,sequence):
    motif,sequence = motif.upper(),sequence.upper()
    start_position = sequence.find(motif)
    if start_position >= 0:
        end_position = sequence.find(motif)+len(motif)
        return (start_position,end_position)
    else:
        print "The motif %s does not exist" % (motif)

motif1 = motif_finder ("GGGTTCACTT",human_sequence)
print motif1,human_sequence[motif1[0]:motif1[1]]

motif2 = motif_finder ("GATCA",human_sequence)
print motif2,human_sequence[motif2[0]:motif2[1]]

motif3 = motif_finder ("AACAT",human_sequence)
print motif3,human_sequence[motif3[0]:motif3[1]]

motif4 = motif_finder ("GGTGTGGGGGG",human_sequence)
print motif4
```

```
(366, 376) GGGTTCACTT
(253, 258) GATCA
(159, 164) AACAT
The motif GGTGTGGGGGG does not exist
None
```

d) Modify the function created in c), so that it can take a variable number of motifs as arguments and return a dictionary with the motifs as *keys* and the tuple with the positions as *values*.

(This is a tough one)

In [44]:
```
def motif_finder2 (sequence,*motifs):
    motifs_dic = {}
    for motif in motifs:
        motif,sequence = motif.upper(),sequence.upper()
        start_position = sequence.find(motif)
        if start_position >= 0:
            end_position = sequence.find(motif)+len(motif)
            motifs_dic[motif] = (start_position,end_position)
        else:
            motifs_dic[motif] = "This motif does not exist"
    return motifs_dic

motifs = motif_finder2 (human_sequence,"GGGTTCACTT","GATCA","AACAT","GGTGTGGGGGG")

for key,value in motifs.items():
    print key, value
```

```
GGGTTCACTT (366, 376)
AACAT (159, 164)
GATCA (253, 258)
GGTGTGGGGGG This motif does not exist
```

Problem 2

The list blast_out from the examples module contains a short table of blast results in a simplified tabular format with the following fields per hit:

query sequence \t ***subject sequence*** \t ***identification percentage*** \t ***e-value***

a) Define a function that returns a list containing only the blast hits with an e-value below 1×10^{-5} . This e-value cut-off should be the default, but the function must be flexible enough to be called with different cut-off values.

Tips:

- Python is able to recognize and interpret numbers in scientific notation when converted with float(), e.g. float(1e-7);
- Define the function using two arguments: one for a list, and the other for the e-value number;
- You can use the .split() method of strings to separated the different field in each blast hit)

In [52]:
```
def blast_parser (blast_list,evalue_threshold=1e-5):
    new_list = []
    for hit in blast_list:
        hit_fields = hit.split("\t")
        evalue = hit_fields[3].strip("\n")
        evalue = float(evalue)
        if evalue <= evalue_threshold:
            new_list.append(hit)
        else:
            pass
    return new_list

Blast_hits_1E7 = blast_parser (blast_out)
print Blast_hits_1E7
```

```
['gnl|SpeciesA|80761\tgnl|SpeciesC|BC1T_16434\t77.78\t3e-10\n', 'gnl|SpeciesA|80761\tgnl|Specie
```

b) Based on the list return by the function in a), define a function that sorts the blast hits according to their identification percentage into three lists of high (100-90%), moderate (90-60%) and low (below 60%) identity percentage. Return those lists into separate variables and calculate the number of hits in each identity percentage class.

Tips:

- Once again you can use the .split() method of strings, but now we are interested in the "identification percentage" field, instead of the "e-value" field);
- Remember that functions can return multiple values (separated by commas)

In [61]:
```
def blast_parser2 (blast_list):
    new_list,high_list,moderate_list,low_list = [],[],[],[]
    for hit in blast_list:
        hit_fields = hit.split("\t")
        id_percentage = hit_fields[2].strip("\n")
        id_percentage = float(id_percentage)
        if id_percentage <= 60:
            low_list.append(hit)
        elif id_percentage > 60 and id_percentage <= 89:
            moderate_list.append(hit)
        elif id_percentage > 90:
            high_list.append(hit)
    return high_list, moderate_list, low_list

high_list, moderate_list, low_list = blast_parser2 (Blast_hits_1E7)
print """There are a total of %s hits in the original dataset:
- %s hits presented high identity scores
- %s hits presented moderate identity scores
- %s hits presented low identity scores
""" % (len(Blast_hits_1E7),len(high_list),len(moderate_list),len(low_list))
```

```
There are a total of 23 hits in the original dataset:
- 12 hits presented high identity scores
- 11 hits presented moderate identity scores
- 0 hits presented low identity scores
```

c) Modify the function in a) so that it can take a file object as input, instead of a list.

In [44]:
```
# First method: Loads the whole file into the memory. Not recommended for very large files.
def blast_parser3 (blast_infile,evalue_threshold=1e-7):
    infile = open(blast_infile)
    blast_list = infile.readlines()
    new_list = []
    for hit in blast_list:
        hit_fields = hit.split("\t")
        evalue = hit_fields[3].strip("\n")
        evalue = float(evalue)
        if evalue =< evalue_threshold:
            new_list.append(hit)
        else:
            pass
    return new_list

#Second method: Loads only one line of the file into the memory each time. Much more memory ef
#only be performed once before the file object is exhausted.
def blast_parser4 (blast_infile,evalue_threshold=1e-7):
    infile = open(blast_infile)
    new_list = []
    for hit in infile:
        hit_fields = hit.split("\t")
        evalue = hit_fields[3].strip("\n")
        evalue = float(evalue)
        if evalue =< evalue_threshold:
            new_list.append(hit)
        else:
            pass
    return new_list
```

c) Open the "blast_out.txt" file in read mode and use the previous function to return a list of blast hits with e-values below 1×10^{-15} .

In [47]:
```
Blast_hits_1E5 = blast_parser3 ("blast_out.txt",evalue_threshold=1e-15)
print Blast_hits_1E5
```

```
['gnl|SpeciesA|56811\tgnl|SpeciesL|1074362\t96.08\t1e-20\n', 'gnl|SpeciesA|56811\tgnl|SpeciesN|
```

d) Create a script that includes the function defined in a). When executed through the terminal, the script should prompt the user for:

- the path and filename of the blast output file that is going to be read
- the desired e-value cutoff
- the name of the output filename

The script must open and read the provided input file, and write all blast hits above the desired e-value cutoff to a new file with the name provided by the user.

Tips:

- Use the raw_input() and input() functions to collect information from the user/keyboard - Their values can be stored in variables to be used latter;

In [50]:
```
#!/usr/bin/python

infile = raw_input("Please provide the path to the input file:\n>")
evalue = input("Please provide the e-value cutoff:\n>")
outfile = raw_input("Pleave provide the name of the output file:\n>")

def blast_parser5 (input_file,evalue_threshold,output_file):
    infile = open(input_file)
    outfile = open(output_file,"w")
    new_list = []
    for hit in infile:
        hit_fields = hit.split("\t")
        evalue = hit_fields[3].strip("\n")
        evalue = float(evalue)
        if evalue =< evalue_threshold:
            outfile.write(hit)
        else:
            pass

blast_parser5 (infile,evalue,outfile)
```

Problem 3:

Use python to open the "My_fasta.fas" file in read mode.

a) Define a function that returns a dictionary with the fasta headers as *keys* and their sequence as the corresponding *value*.

Tips:

- For this exercise, we ran out of tips. Sorry.

In [21]:
```
def fasta_parser (fasta_file):
    fasta_dic = {}
    infile = open(fasta_file)
    for line in infile:
        if line.startswith(">"):
            header = line[1:].strip("\n")
            fasta_dic[header] = ""
        else:
            fasta_dic[header] += line.strip("\n")
    return fasta_dic

fasta_dic = fasta_parser ("My_fasta.fas")
```

b) Define a function that performs some quality checks for each sequence. Check if:

- all sequences are of the same size. If not, the function should print a message informing which taxa have sequences of different length.
- there are no illegal characters in each sequence. If there are, the function should print a message informing which taxon's sequence has problems and what is the illegal character. (Tip: Use the *IUPAC_codes* list from the *data* module to check for illegal characters)

In [23]:
```
def quality_check (fasta_dic):
    sequence_sizes = []
    for taxon, sequence in fasta_dic.items():
        if sequence_sizes == [] and len(sequence) not in sequence_sizes:
            sequence_sizes.append(len(sequence))
        elif sequence_sizes  != [] and len(sequence) not in sequence_sizes:
            sequence_sizes.append(len(sequence))
            print "The taxon %s has a different sequence size: %s" % (taxon,len(sequence))
        for nucleotide in sequence:
            if nucleotide not in IUPAC_codes:
                print "The taxon %s has an illegal character: %s" % (taxon,nucleotide)

quality_check (fasta_dic)
```

```
The taxon Clupus_63137 has a different sequence size: 610
```

c) Create a function that uses the dictionary created in a) and collapses taxa with identical sequences into the same haplotype. Write these unique haplotypes to a new file (My_fasta_collapsed.fas) in fasta format, and write the correspondance between haplotype and taxon name in another file (My_haplotype_list.txt).

In [35]:
```
def collapse_sequences (fasta_dic):
    Collapsed_dic = {}
    for taxon, sequence in fasta_dic.items():
        if sequence in Collapsed_dic:
            Collapsed_dic[sequence] += "%s; " % (taxon)
        else:
            Collapsed_dic[sequence] = taxon+"; "
    return Collapsed_dic

def file_writer (sequence_dic,sequence_outfile,haplotype_outfile):
    outfile_fasta = open(sequence_outfile,"w")
    outfile_haplotypes = open(haplotype_outfile,"w")
    Haplotype = 1
    for sequence, taxa in sequence_dic.items():
        outfile_fasta.write(">Haplotype%s\n%s\n" % (Haplotype,sequence))
        outfile_haplotypes.write("Haplotype %s: %s \n)" % (Haplotype,taxa))
        Haplotype += 1

collapsed_dic = collapse_sequences (fasta_dic)
file_writer (collapsed_dic,"My_fasta_collapsed.fas","My_haplotype_list.txt")
```