

Hands-On Lab

Module 08: Sho

Lab version: 1.0.0

Last updated: 12/6/2011

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

© 2011 Microsoft Corporation. All rights reserved.

Microsoft, MS, Windows, MSDN, Visual Basic, Visual C++, Visual C#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

[Creative Commons License](http://creativecommons.org/licenses/by/3.0/)Microsoft Biology Initiative Training by [Microsoft Corporation](http://research.microsoft.com/bio) is licensed under a [Creative Commons Attribution 3.0 Unported License](http://creativecommons.org/licenses/by/3.0/). Based on a work at [research.microsoft.com/bio](http://research.microsoft.com/bio).

[Module 08: Sho 1](#_Toc310783208)

[Using Sho to analyze biological data 4](#_Toc310783209)

[Objectives 4](#_Toc310783210)

[System Requirements 4](#_Toc310783211)

[Task 1: Installing Sho and IronPython 5](#_Toc310783212)

[Task 2: Loading .NET Bio into Sho 7](#_Toc310783213)

[Task 3: Calculating G/C content 9](#_Toc310783214)

[Task 4: Read Trimming 12](#_Toc310783215)

Using Sho to analyze biological data

This Hands-On Lab walks you through using the Sho data analysis tool with the .NET Bio platform to load and analyze a set of sequences.

Objectives

In this Hands-On Lab, you will get some experience building python scripts that uses Sho and .NET Bio

* + Adding references to .NET Bio in the Sho tool.
  + Loading a set of sequences into Sho.
  + Culling a data set from the sequences.
  + Displaying graphical representations of the results.

# System Requirements

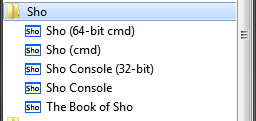
You must have the following items to complete this lab:

* + Microsoft Visual Studio 2010
  + .NET Bio 1.0 or later
  + Iron Python 2.7 or later
  + Sho 2.0 or later
  + Windows XP SP3 or better

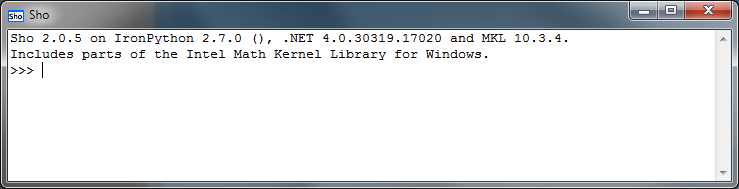
Task 1: Installing Sho and IronPython

In this task we will ensure your environment is properly setup with Sho and IronPython. If you already have Sho and IronPython installed, then you can skip this task and move onto Task 2 where you will begin to use the tool to analyze some bio data.

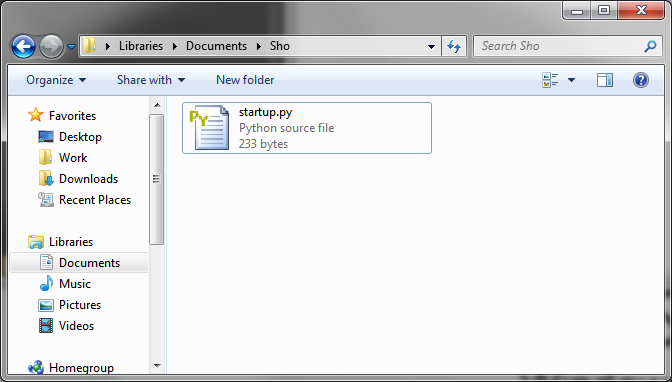
1. As a first step, download IronPython 2.7.1 (or later) from <http://ironpython.net> and install it.
2. Next, download Sho 2.05 from <http://research.microsoft.com/en-us/projects/sho> and install it onto your machine. The order you install is significant.
3. Once Sho is installed, you should have a new group in your start menu (the following is a screen shot from Windows 7, earlier versions of Windows may look slightly different):



1. Go ahead and run the Sho Console – this will be 32-bit or 64-bit based on your machine configuration (although as with any bio-related work, we recommend a 64-bit operating system).
2. You should see the following prompt:



1. If the versions do not match and are *earlier* than what is shown above, then double check your installation to ensure you are running on the proper version of IronPython and Sho.
2. Close the Sho window.
3. As a last step, add the IronPython libraries into the Sho command search path – you can either do this each time you start the session by appending to the **sys.path** variable, or do it once and for all by adding it to your Startup.py python script in your documents folder (it should be in your user Documents folder in a sub-folder called “Sho”):



1. Edit the startup.py file and add the following line to the end: (note: this is for a 64-bit install, you might need to alter the path slightly if you customized your IronPython install, or it’s in a different location that shown below).
   1. sys.path.append(r"C:\Program Files (x86)\IronPython 2.7\Lib")
2. Save and close the file.
3. Open the Sho console again and type:
   1. >>> import collections
4. If it succeeds without an error, you have successfully added the path. If it fails, double-check the path to the libraries (ensure it exists in explorer for example) and correct it if it’s wrong.

Task 2: Loading .NET Bio into Sho

In this task we will begin to use the Sho tool to load some biological data and do some simple analysis.

1. Start the Sho interactive console.
2. As a first step, let’s load the .NET Bio core assembly (bio.dll). It is located in the **%Program Files%\.NET Bio\1.0\Tools\bin** folder, so for a 64-bit machine that would be:
   1. C:\Program Files (x86)\.NET Bio\1.0\Tools\Bin
3. To load the assembly, let’s use the **ShoLoadAssembly** function – this combines the IronPython **clr.AddReference** with **clr.LoadAssembly** into a single step. The single expected parameter is the path and filename of the assembly to load. Remember to either escape the back-slashes, use forward slashes, or prefix the string with “r” to get IronPython to escape the string for you.
   1. >>> ShoLoadAssembly(r"C:\Program Files (x86)\.NET Bio\1.0\Tools\Bin\bio.dll")
   2. <Assembly Bio, Version=1.0.4300.28638, Culture=neutral, PublicKeyToken=13335e9abfbf69d0>
   3. >>>
4. Once bio.dll is loaded, we can import the necessary namespaces we want to work with. Go ahead and import the core types in **Bio** and **Bio.IO.FastA**.
   1. >>> from Bio import \*
   2. >>> from Bio.IO.FastA import \*
   3. >>>
5. Next, we have some sample data in the **Data** folder included with this lab. Go ahead and load the sequences contained in the **454ReadsRaw.C1.fasta** file. To do this, we need to create an instance of the **FastAParser** (passing the filename), and then call the **Parse** method – for convenience, you can wrap the returned **IEnumerable<ISequence>** into a python list.
   1. Name the variable holding the FastAParser “parser”.
   2. Name the variable holding the returned sequences “allSequences”.
6. If you need help, here’s the sequence of commands you need to type:
   1. >>> parser = FastAParser(r"Y:\Dropbox\Business\Work\Microsoft\MSR\NET Bio V1\Module 08\Lab\Data\454ReadsRaw.C1.fasta")
   2. >>> allSequences = list(parser.Parse())
   3. >>>
7. If it all goes well, you should be able to interrogate your sequence variable and get a set of sequences back on the console – just type “allSequences” and press ENTER. This will invoke the **ToString()** override for each sequence – showing a section of the loaded data as a string.
8. For more structured output, let’s take one sequence and print out the ID, count of elements along with a string containing the characters.
   1. Assign the single sequence to a variable named “sequence”
   2. Use the first sequence in the list (index position [0])
   3. For the string contents, use the built-in **Sequence.ConvertToString** method – this is actually on the Sequence type, but not exposed on the interface we have. Here, the dynamic typing nature of Iron Python come in handy because if the method exists on the underlying object, it will be callable!
   4. **ConvertToString** takes a starting position and count to create a string from.
9. If you need some help, here’s an example:
   1. >>> sequence = allSequences[0]
   2. >>> print sequence.ID + " has " + sequence.Count.ToString() + " elements, " + sequence.ConvertToString(0,sequence.Count)
   3. GYO778V02FSC2M has 932 elements, gactacacgtagtatGCTAGAATATCAAAAATTTTGGATTAAATTTTTATGTTAACTTTTATAGTGTTTAGTTCAAATTTTTATATTCTTATTGTTCCATTATTAAATAGTATGTAATTTTGAAACATACagtacgcaaggggaaaaaaattagtaattaggctcattgagcattataagtaaacatttaaaactaatgatgggtgcaaaggagtaggaggtaattaagttagaacttaaacttttagttacgaaaactaaaaaatttaaaaattttaagtaagtttaaacctttaaaaacttaaaaaatttaaaaattaaaggtagttaaactttaaattaaaagtaacgtaattaacgggttttacccgggttaaaccttactttatttttacggtttttaacgttaacccttaacccttaattttacgtaaccgtactttaattttaaagtaagtaggttagggttaaattaagtaggttagttacgtactaccttacttaaccgtacctaacttactttaaccttaactactaaccttaaccgtctaaacccaaaccaccttagtttacacctaacctctaccaacaacgtacgtacccttaccttttattttaagttagtttttacttttttcttaataggttacactcgtacaagtaaggttagagggtactcttttccaactaacttacctttaatttttacgtagtaccctttaaaaacctttaaaaaggttacggaaccttaaaataccgtaaggttaaaaacgttttaaaaccgtttaaaagtaaaaaacttaaacgttaaaacttaaaggtttagggaacttacgaaacccggtttttaacccctttttatttacgttaaaacctttaaaagttagggtaaacggttaagtaaaacgtaaaaatttaaccggtttaagggg
   4. >>>

10. As a last step, try turning the above code into a function definition and invoking it for all sequences in the list. The following is one way to do that.

* 1. >>> def dumpSequence(sequence) :
  2. print sequence.ID + " has " + sequence.Count.ToString() + " elements"
  3. print sequence.ConvertToString(0,sequence.Count)

  6. >>> dumpSequence(allSequences[1])
  7. GYO778V02HCJGT has 593 elements
  8. gactacacgtagtatATAATTGATAACTTAATTGCAAAATAATTTAAATGAATTTACAAAATTTAAATAAATGGATTATAAATAGATAATTGGGTTACCCAATTCATTTTTTGACTTACCCATTTTATACGC ...
  9. >>> for s in allSequences : dumpSequence(s)
  10. ...

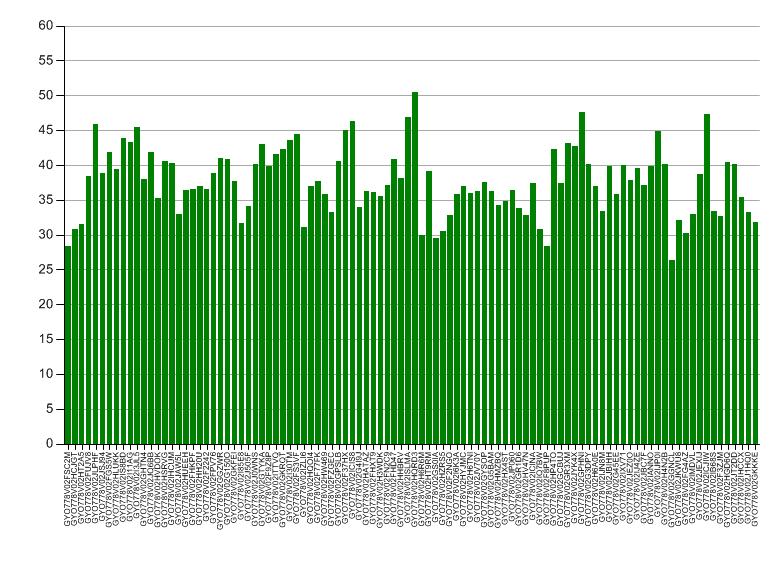
Task 3: Calculating G/C content

In this task we will use Sho to calculate the G/C content ratio for a set of sequences and then visualize it with a set of graphs.

1. Start the Sho interactive console.
2. Load the .NET Bio core assembly. (See Task 2 if you need some help here).
3. Load the sample **454ReadsRaw.C1.fasta** from the data folder, again – see Task 2 if you need some help here.
4. The G/C content ratio is simply a measure of how many G+C elements there are compared to the overall sequence. It is fundamentally calculated with the following formula:

**( G + C ) / (G+C+A+T)**

1. Here, we will define a new function to calculate the above ratio and return it as a float. A few things to think about:
   1. Many sequences use letter casing to designate some property of the monomer at that position – i.e. it might represent certainty or some secondary validation of the value if it’s upper case vs. lower case. We need to capture both so we need to test for both.
   2. We are going to ignore gaps in the sequence (if any exist)
   3. We need to check to make sure the entire sequence isn’t composed of gaps since that’s the divisor here.
2. We could use the built in **SequenceStatistics** class to get some of this information, but where possible, it’s handy to use built-in Python capabilities. In this case, Python 2.7 includes a handy **Counter** class in the collections module. This counts the number of unique elements and places them into a bucket (much like **SequenceStatistics** actually). Use the intellisense support in the editor to get a sense of how to use it, or use **dir** or **help**.
   1. The Counter class is in the collections module – you will need to import that first.
   2. If the import fails, double-check that you have added the appropriate path per Task 1.
3. Once you are finished defining the function, call it for the 1st sequence, you should get back: 0.2832618025751073, or 28.32%
4. Here is an example definition of the function if you need some help:
   1. def calcGCratio(seq) :
   2. table = Counter([chr(c) for c in seq])
   3. G = table['g'] + table['G']
   4. C = table['c'] + table['C']
   5. A = table['a'] + table['A']
   6. T = table['t'] + table['T']
   7. total = G + C + A + T
   8. if total > 0 : GC = (float(G) + float(C)) / total
   9. else : GC = float(0)
   10. return GC
5. Once you have the function defined, try enumerating through all the sequences and dump out the G/C ration for each – you can use a format string of **'{:.2%}'** to get a percent with two decimal places.
6. Next, use your function to generate a collection of values as percentages (multiply the result of each value by 100) and then graph that with the ID using the bar function of Sho for the first 100 sequences in the data.
   1. To constrain the data, use the **range** function and index into your allSequences list.
   2. >>> vals = [calcGCratio(allSequences[i])\*100 for i in range(0,100)]
   3. >>> names = [allSequences[i].ID for i in range(0,100)]
   4. >>> bar(names,vals,'g')
7. Finally, save the bar graph to disk using the **saveplot** function while the bar is showing on the screen. You can just type in **saveplot** with a filename – it works against the active graph. Use the **dir** or **help** method to see parameters.
8. Your graph should look something like:

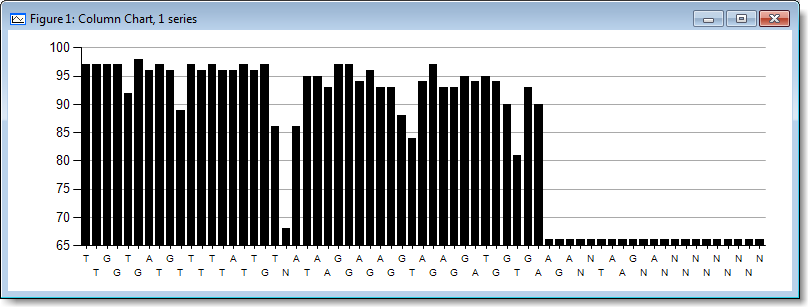


1. Using the graph you can get a quick sense of which sequences fall below thresholds of G/C counts. This can be particularly helpful when you are looking for patterns across organisms (here it’s all the same organism so the data isn’t particularly compelling).
2. As a last step, try putting the entire data set into a data grid – if you want to show the sequence ID as a column, you will need to move the data into an ObjArray with a loop. One solution is presented below:
   1. # Show id + values in grid
   2. oa = ObjArray(len(names),2)
   3. >>> for i in range(len(names)) :
   4. oa[i,0] = names[i]
   5. oa[i,1] = vals[i]
   7. >>> dgv(oa,cols=['ID','G/C Count'])

Task 4: Read Trimming

In this task we will use Sho to trim the reads of a FastQ input file by looking at the quality scores present on the sequence. You will then write the updated sequences back out to a FastQ file.

1. Start the Sho interactive console.
2. Load the .NET Bio core assembly. (See Task 2 if you need some help here).
3. Load the sample **1\_sequence.fastq** from the data folder. Here you will need to import the FastQ namespace from Bio.IO and use the FastQParser. If you need help, Task 2 is a good guide.
4. There is a single sequence in this file – assign that single sequence a variable name (we use **seq** below), and dump out the **QualityScores** property to see what it consists of.
   1. You will find it is a Byte array where each byte represents the score one nucleotide in the sequence.
   2. The sequence comes from an Illumina sequencer (check the **FormatType** property to see this).
5. Our goal is to trim the start and end of the sequence based on a particular range – i.e. we want to create a new sequence that consists only of elements within our range, but once we hit the first element, take the remainder and then do the same for the reverse of the sequence so we end up with the start and end reads trimmed off.
6. Define a function which is passed a sequence and a minimum score. It should perform the following action:
   1. Name the function **TrimReads**.
   2. Return a tuple containing the sequence data and quality score that matches our criteria
   3. Enumerate the passed sequence and remove all elements from the from and end which do not meet the minimum passed quality score.
   4. **Hint**: remember you can index to the *end of an array* using the (-1) indexer in Python
7. Here is an example if you need some help – note that there are many ways to do this, the following while not the most efficient, is probably the simplest.
   1. def TrimReads(seq, minValue) :
   2. out = [chr(v) for v in seq]
   3. qs = [float(v) for v in seq.QualityScores]
   4. while(qs[0] < minValue) :
   5. out.pop(0)
   6. qs.pop(0)
   7. while(qs[-1] < minValue) :
   8. out.pop(-1)
   9. qs.pop(-1)
   10. return (out,qs)
8. Run your function on the loaded sequence to see the output – for example, passing 50 as the minimum, you should get something like:
   1. >>> out = TrimReads(seq,50)
   2. >>> out
   3. (['T', 'T', 'G', 'G', 'T', 'G', 'A', 'T', 'G', 'T', 'T', 'T', 'T', 'T', 'A', 'T', 'T', 'G', 'T', 'N', 'A', 'T', 'A', 'A', 'G', 'G', 'A', 'G', 'A', 'G', 'G', 'T', 'A', 'G', 'A', 'G', 'G', 'A', 'T', 'G', 'G', 'T', 'G', 'A', 'A', 'G', 'A', 'N', 'N', 'T', 'A', 'A', 'G', 'N', 'A', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N'], [97.0, 97.0, 97.0, 97.0, 92.0, 98.0, 96.0, 97.0, 96.0, 89.0, 97.0, 96.0, 97.0, 96.0, 96.0, 97.0, 96.0, 97.0, 86.0, 68.0, 86.0, 95.0, 95.0, 93.0, 97.0, 97.0, 94.0, 96.0, 93.0, 93.0, 88.0, 84.0, 94.0, 97.0, 93.0, 93.0, 95.0, 94.0, 95.0, 94.0, 90.0, 81.0, 93.0, 90.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0, 66.0])  
      >>>
9. You can put it into a bar graph if you like:



1. Now, let’s save the data back out – to do that, we have to coerce this data into a form that .NET Bio can work with. First, we need to move the two elements in the tuple into Byte array types (real .NET **byte[]** elements).
   1. Import all the types in the **System** namespace (or just **Byte** if you like).
   2. Next, create a **byte[]** using the IronPython syntax “**Array[Byte]**”
   3. Pass the byte values representing the nucleotides to the constructor of the byte array – this will fill the contents, you can use the **ord()** function to convert each character to the byte value.
   4. Assign the resulting array to the variable **data**.
   5. >>> data = Array[Byte]([ord(v) for v in out[0]])
2. Do the same steps to get a byte array of the quality scores (the second item in the tuple).
   1. Assign the resulting array to the variable **scores**.
   2. Since these were already bytes, no cast will be necessary here.
   3. >>> scores = Array[Byte]([v for v in out[1]])
3. Next, create a new **QualitativeSequence**, passing the data.
   1. It needs an alphabet – use the existing sequence’s Alphabet property.
   2. It also needs a **FormatType** – again, rely on the existing sequence’s format type.
   3. Assign it to the variable **qs**.
   4. qs = Bio.QualitativeSequence(seq.Alphabet,seq.FormatType,data,scores)
4. Finally, write the data out using a FastQFormatter.
   1. Create a formatter with a unique filename
   2. Write the sequence
   3. Close the formatter
   4. Verify the file was written.
   5. >>> formatter = FastQ.FastQFormatter("out.fastq")
   6. >>> formatter.Write(qs)
   7. >>> formatter.Close()

This concludes the lab, feel free to play with Sho and see what other kinds of analysis you can do on the provided sequences, or even other sequences. Have fun!