

Python programming for Life Science researchers

An Introduction to Biopython

Sebastian Bassi

Universidad Nacional de Quilmes, Argentina

Outline

Why is this tutorial important?

Python is a popular computer language that is gaining momentum between scientific users. Python is used in a wide range of applications, from P2P (like Bittorrent) to dynamic generated webpages (like Google uses). Scripting languages like Perl, Python and Ruby are used extensively for data manipulation, a task frequently used in scientific work. BioPython will be used as example of the power and simplicity of Python, since you can for example do a BLAST search using a one line of code and process it with very few lines. Another advantage worth pointing is that Python programs runs on virtually every platform, from PDA to supercomputers. After tutorial completion, students should be able to make their own programs.

What will it cover?

Concepts

Python concepts. What is different from other languages.

Interactive use. Python as a calculator.

Data structures. Variables, Lists, strings, dictionaries

Program control flow. For, if-else, while

Modularize. Functions. Modules.

File management. Reading and writing text files. Data manipulation

XML. Overview and processing.

Useful modules. Cgi, htmlgen, BioPython. Build a melting point calculator with a web interface that generates HTML and use BioPython functions.

CONTENTS

INTRODUCTION.....	5
THE INTERACTIVE INTERPRETER.....	6
PYTHON AS A CALCULATOR.....	6
DATA TYPES.....	6
STRING DATATYPE.....	6
LIST DATATYPE.....	8
LIST NOTATION.....	8
LIST OPERATIONS.....	8
TUPLES.....	9
DICTIONARIES.....	10
PROGRAM FLOW.....	11
FUNCTIONS.....	12
MODULES.....	13
READING TEXT FILES.....	13
WRITE TEXT FILES.....	14
DATA MANIPULATION.....	14
XML: BASIC OVERVIEW	16
XML: SOME REAL WORLD SAMPLES	17

XML: ANOTHER SAMPLE, BLAST.....	17
XML: ANOTHER SAMPLE, SVG.....	17
XML: PARSER WITH ELEMENTTREE.....	17
WHAT IS BIOPYTHON.....	18
BIOPYTHON SAMPLE.....	19
MELTING POINT CALCULATOR.....	23
GENERATE TM IN HTML FROM MULTIPLE SEQUENCES USING PYTHON.....	25

Introduction

Why Python?

Let me introduce Python first. Python is a modern programming language developed in the early 1990s by Guido Van Rossum. There are many characteristics of this language that is worth point out:

Easy to read: A program written in Python is easy to read, for another programmer or even for the same programmer some time after the code was written. The same can't be said for all languages out there. Sometimes is so easy to read that the code resemble the pseudocode used in many books.

Easy/Fast to write: Python handles most low level function, so you don't have to worry about pointers and memory allocation. This will allow you to focus just in your work. This also result in faster times for coding. With current computer speeds, most of the time is more important the coding time than program running time (this of course depends on the task).

Batteries included: Python has many functions built-in. Other languages depends of several external libraries to accomplish the same task that can be done with a standard Python installation.

Multiplatform: There are version of Python for all popular computing platform (like Windows, Linux, Mac, Solaris and even some phones and PDAs). This make your programs portable between these platforms.

Dynamically typed: It doesn't use explicit datatype declarations. When you use a variable for first time, is automatically declared.

Strongly typed: Once a variable is defined, it remains on it type unless explicit declared. If you have an integer, you can't use it as a string without converting it.

Friendly community: This is not a feature built into the language, but it makes a good point you should watch for before deciding to “adopt” a language.

The interactive interpreter

Python has two modes of use: Interactive and batch mode. In the interactive mode you write any command and then you get a reply from the interpreter after you press enter. Is the ideal way to test and try how Python works. This screen shows Python version and platform where the interpreter is installed. Version number is important since there are many improvements on each version.

Most of the time you will use the batch mode, but even then you could use interactive mode for debugging porpoises.

Python as a calculator

In the interactive mode you can write expressions and get answers in real time. On this slide you could see that integer values are integer unless you declare them as float. More about data type will be introduced in the next slide. The underscore (_) is the last output of the interpreter.

Data Types:

There are different data types, in this slide there are most types used for numbers:

Integer: Integer number up to 2.10^{31} .

Long: Integer larger than 2.10^{31} . The upper limit depends on hardware architecture.

Float: Floating point numbers.

Type command return the data type.

String datatype:

String are limited by quotes, double quotes or even triple quotes. Single and Double quote can be used in alternate way, as long as you start and end with the same type of quotes. Triple quote are used to preserve multiple line format.

You can use “slice notation” to refer to a substring. Slice notation will be explained in list data type. With plus (+) you concatenate two or more strings, but you can't concatenate strings with numbers. With str() you can convert any other data type to string.

Special characters: You can escape non-printable characters using “\”. Here is a list of supported escape sequences:

\'	Single-quote
\"	Double-quote
\\	backslash
\a	bell
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab

Join multiple strings with string.join:

```
string.join(["This is a","test"])
```

This is a test

You can join using other characters than spaces, like “;”:

```
string.join(["This is a","test"],";")
```

This;is;a;test

Split function is very useful for file parsing:

```
string.split("Another test"," ")
```

```
["Another","test"]
```

Converting string to numbers:

```
string.atof("444.76")
```

List datatype:

A list is a set of data under the same variable name. Are like C vectors or VB and Perl arrays. A list can contain different types of elements, like int, string and another list (nested list). The first element of a list is the element 0, the second is the number 1 and so on. These numbers are called list index. To access inside a element inside a list, you need to use another index. This slide shows how to access a element in a string element inside a list.

List notation:

For operations involving only one element, you can think the index as a particular element of the list. When you want to access to a group of elements, you should think of list indices as positions between elements. On the slide there are some usage sample. When you leave one index as blank a 0 value is assumed. To count from the last element, negative indices are used.

List operations:

There are three ways to add data into a list.

Append: Add elements into the last position

Insert: Add elements in any arbitrary position

Extend: Add a list to the last position.

List: Delete elements

There are two ways to remove elements:

pop(n) will retrieve the nth element of the list. The value can be stored in a variable (or printed, or handled in any way).

remove(n) will remove the nth element of the list, but without retrieving the value.

There are more useful methods in the lists:

List membership: in and not in can be used to verify if a specific element is inside the list.

```
6 in [34,5,6,7]
```

```
1
```

```
9 not in [1,2,6,9]
```

```
0
```

List initialization with *:

```
f = [4] * 4
```

```
f
```

```
[4,4,4,4]
```

List minimum and maximum:

```
max(f)
```

```
4
```

```
min([3,4,7,5])
```

```
3
```

Count elements in a list:

```
g=[3,4,5,6,7,8,7,6,5,7]
```

```
g.count(7)
```

```
3
```

```
g.count(9)
```

```
0
```

Search inside a list: Index

```
g.index(3)
```

```
1
```

An error will be raised if the value is not in the list. It is recommended to execute a in operator before an index.

Tuples:

This is a data structure very similar to lists, but can't be modified. They are immutable.

The advantage is that they are faster to operate than lists. They are "write protected" lists.

Tuples can be used to make several assignments at the same time:

```
var1, anothervar, var3 = 56,34,89
```

This line replaces 3 lines:

```
var1 = 56
```

```
anothervar = 34
```

```
var3 = 89
```

Converting between tuples and lists:

List function produce a new list with the same elements as the input. Tuple function does the opposite.

```
list((3,4,6,8))
```

```
[3,4,6,8]
```

```
tuple([5,8,8,7])
```

```
(5,8,8,7).
```

Dictionaries:

This is a kind of “associative arrays”. That is, it stores one to one relationship between keys and values.

A dictionary can be created empty:

```
k={}
```

Or with data:

```
k["X"]=242.6
```

```
k["T"]=178.3
```

Elements in a dictionary has no implicit order (that is different from lists).

Some methods of Dictionaries:

Len returns the number of elements in the dictionary:

```
len(k)  
2
```

keys will return a list of all keys in a dictionary:

```
k.keys()  
["X","T"]
```

Values will return a list with the values stored in a dictionary:

```
k.values()  
[242.6,178,3]
```

Has_key will test if there a specific key:

```
k.has_key("X")  
1  
k.has_key("K")  
0
```

Del will remove an entry from the dictionary:

```
del k["X"]  
k.items()  
[("T",178.3)]
```

Program flow:

The first control structure we will see is "If else". It will test for a condition, if it is true, the program will execute the code that is under the if. If the condition is not true, the else part will be executed. In this slide we also introduce the indentation. That is, the space you leave to indicate a block of code. In this case, this code is checking for the size of the variable length is greater than 20. In this case, it will display a

sequence with the “...” string in the middle. Otherwise (else), it will just display the whole seq variable (since it is smaller than 20 chars, we are sure that it will fit in the output space).

In order to evaluate multiple conditions, there is the elif clause. It works as shown in the slide (like case in C).

for: To iterate inside a list. In the slide there is a list called output_parts with three small strings. In this case, opart will take the value of each element in the list.

If you want to iterate in a list of numbers (“a la Basic”), just create a list of number with range function.

```
range(5)  
[0,1,2,3,4]
```

while: To execute a code as long as a condition is true.

Functions:

A standard way to modularize your code is to create functions. A function is a portion of code that can be accessed from any other part of the program. In Python, you use “def” to define a function, a name, and enclose between parenthesis all the variables that the function needs. To assign a default value, just use “=” in the definition. Like this:

```
def a_new_function(var=value)
```

To return a value to the program, use “return”. If the function doesn't return anything, you can write it as return None.

Only one value can be returned from a function. To return more than value, you can return a list containing multiple values.

Modules:

In Python you can store functions, constants and dictionaries in a file called module. You can invoke modules from a program or from interactive mode. Python provides several modules and there are many more that can be downloaded from the Internet. In the slide there is a sample usage of the modules.

To bring a function from a module you have to import it first. To import the math module, just do: `import math`

We will import several modules during this tutorial.

Reading text files:

Reading a text file in python is a three step process.

1: Open the file and assign it a handle.

```
HandleName=open("PATHNAME","r")
```

On the open function the first term is the filename, with path (if no path is specified, current working directory is assumed). the second term is the first letter of the open mode, that is read, write and append.

the handlename is used from this point on, every time you refer to the file, instead of using the filename, just use the handle.

If you print that handle, you will see only an hexadecimal code corresponding to the handle. To see the actual contents of the file, you need the step two:

2: Read the file. There are several functions to read the contents of a file:

`read(n)=` Will read the first n characters of a file. Without arguments, will read the whole file.

`readline(n)=` Will read the line number n. If you invoke this function without argument, it will read the first line the first time you call it. Next time will read the second. When there is nothing more to read from a file, it will return an empty string

`readlines()`= Will return a list of string with all lines in the file.

End of line (EOL) code is determined based on host operating system.

3: Close the file:

```
HandleName.close()
```

Python can close the file automatically when the program ends.

Write text files:

Writing a file is very similar to reading a file. Step one and three are the same. The main difference is in step two. Instead of reading a file, you have to write into the handleName, like this:

```
HandleName.write("This will go into a text file")
```

Data manipulation:

The problem: A text file with data on it should be parsed, that is, read and interpreted by the program, and then display or store only selected information.

Python tools:

- Build-in open file function.
- Control flow structures.
- String manipulation methods.

The slide 22 will introduce a typical data manipulation problem. Here is the output of a BLAST search (in hit table format). The aim is the get to Gene ID (GID) of the genes with a BLAST match better than 45%. With the GID, make the URL to get the sequences.

Line by line explanation of the source code (slide 23):

import string

To make available the string module (that will be use in line 6).

baseurl="http://..."

A string type variable that will be used at line 11. The backslash ('\') at the right allows to split the line into multiple lines as if were a single line.

infile=open('allans.txt','r')

infile is the file handle (see slide 19 for more information on filehandles), this will open the file allans.txt for reading. Allans.txt is the file where the BLAST output is stored.

for line in infile:

This will read all lines in file with "infile" handle.

ids=string.split(line,'\t')[1]

String.split will separate the contents of line (that is, the line that is being reading) and put all different content in a list. To retrieve only one element, it is indicated with [1], this will store only the element 1 of the list (that is, the second element), into the ids variable. The '\t', that represent the tab character, is the data field separator on the hit table. The second element is the one that start with gi|26249933|ref|.

p_ident=float(string.split(line,'\t')[2])

As line line before this one, this will retrieve one element (the 3rd). The float function will convert this string into a float number.

if p_ident > 45:

Since I want to get the GID of the results with a match better than 45%, I have to test for this condition.

#retrieve only gi from ids

This is a comment. This line will not be interpreted by Python.

gi=string.split(ids,')[1]

To get the GID, we extract it from ids variable (see explanation for line 6), This time we split that variable using '[' as a separator. We retrieve only the second element (the GID).

print baseurl+str(gi)

This line should be self-explanatory. We are printing the variable we created in line 2 plus the gi. This way we are printing what we are looking for.

else:

Is used to indicate the beginning of an alternative block, this will be executed only if the "if test" fails, that is, when p_ident is 45 or less.

pass

Its like a dummy statement, without any effect on the code, we use it because a statement is necessary after an else.

XML: Basic Overview

Language to describe data (with no information about data presentation).

Based on text format (binary XML is out of the scope of this tutorial).

XML are "human-legible" (kind of)

Easy to write programs to process XML documents

Header with parsing information:

<?xml version="1.0"?>

Body:

<tagname attribute_name="attribute_value">a text</tagname>

<line type='demo'>A simple line</line>

**Empty element: **

XML: Some real world samples

This is the source of an RSS feed. RSS is XML formatted.

XML: Another sample, BLAST

BLAST can output as XML if commanded to do so. The tags are very self-explanatory.

XML: Another sample, SVG

SVG: Scalable Vector Graphics. SVG defines graphics in XML format. There are SVG standalone viewers, SVG plugins for browsers, and Firefox display SVG files since version 1.5.

XML: Parser with elementtree

There are several parser in Python. Default parser are SAX and DOM. To use both of them you need to know object orienting programming, that is out of the scope of this tutorial. That is why we are using "ElementTree".

Line by line explanation of the source code:

```
from elementtree.ElementTree import ElementTree
```

On this line we retrieve the desired function from the module.

```
myxml=open("slashdot.xml","r")
```

myxml is the handle for the file we want to parse.

```
root=ElementTree(file=myxml)
```

On this line, the ElementTree function is in action. It only needs as argument the filehandle of the xml file. The whole tree is stored in an object called root.

iter=root.getiterator()

To iterate over the file, we need the getiterator function. It returns all elements into iter variable.

for ele in iter:

This line will start to cycle over all the elements in iter.

if ele.keys():

this will check for the presence of keys inside ele

print ele.items()[0][0]

The keys will be printed

else:

If there is no keys inside ele

print ele.tag

Print the tag of each element

print ele.text

Print the text of the associated tag

What is Biopython?

It is a distributed collaborative effort to develop Python libraries and applications which address the needs of current and future work in bioinformatics.

It provides:

- Tools for working with sequences (aa and nt).
- Parsers of all popular bio file formats (fasta, gb, pdb, BLAST output).

- Data retrieve from biological databases.
- Wrapper to bio-programs (BLAST, ClustalW, EMBOSS, Primer3, and more).
- Biological functions like LCC, restriction enzymes cutting, and more.
- Tables and constants.

Biopython sample. BLAST output parsing for vector removing from DNA sequences

This is a BLAST output of a search for a vector inside newly sequenced data. Cloning vector (called pBlueSKP in the table) is found on each sequence. We have to write a script to remove the cloning vector from all our sequences. These sequences are stored in the same directory and the file name correspond with the one on the BLAST output file.

The program will be introduced in two parts, a BLAST parser and a sequence writer

Blast parser:

On the first five lines the script imports all necessary modules.

arin=open("resultado2.txt","r")

arin is the filehandle for the file where the BLAST output file.

seqab1={}

seqab1 is an empty dictionary. Will be used to store the name of the sequence (as key) and a list with start and end position of the match with the vector (as value).

for line in arin:

To walk arin line by line.

if ".ab1" in line and "#" not in line:

This check for the filenames ended in ".abi1" and not inside a comment line

```
nomtemp=line.split("\t")[1]
```

To store in nomtemp the filename, that is the second column on the input line

```
#print nomtempo
```

A comment line with no operational value. It can be safely deleted.

```
if nomtemp not in seqab1.keys()
```

Check whether the name is not in the dictionary (seqab1)

```
seqab1[nomtemp]=[int(line.split("\t")[8]),int(line.split("\t")[9])]
```

Since the name is not in the dictionary, this line will create the dictionary entry for this file (nomtemp as key). The value is the two element list indicated before.

```
else:
```

The next block is executed when the name is in the keys.

```
if seqab1[nomtemp]==[int(line.split("\t")[8]),int(line.split("\t")[9])]:
```

To check if this filename is associated with the same list

```
print "repetido: "+nomtemp
```

In this case, it print it to the screen just to tell me there is a repeated entry.

```
else:
```

The next block is executed when the filename is not associated with this list.

```
if int(line.split("\t")[9])>int(line.split("\t")[8]):
```

Check if the end position is greater than the start position. In this case we should also add this list.

```
listemp=seqab1[nomtemp]
```

```
listemp.extend([int(line.split("\t")[8]),int(line.split("\t")[9])])
```

```
seqab1[nomtemp]=listemp
```

This will expand the list inside the dictionary.

```
else:
```

The next block is executed when the start position is greater than the end position.

```
listemp=seqab1[nomtemp]  
listemp.extend([int(line.split("\t")[9]),int(line.split("\t")[8])])  
seqab1[nomtemp]=listemp
```

This will expand the list inside the dictionary, with the position inverted.

else:

The next block (a pass statement) will be executed when there is no ab1 file, to do nothing.

arin.close()

After parsing all the text file, we close it since this filehandle won't be used again.

for x in seqab1:

To cycle inside all elements in seqab1 dictionary.

arin=open("renames/"+x[:-3]+"txt","r")

arin filehandle is open with files called like the keys in the dictionary. These are the original sequence files (with vector pollution).

parser = Fasta.RecordParser()

A parser is defined, this is a special parser provided by Biopython to read fasta files.

iterator=Fasta.Iterator(arin,parser)

A iterator is used to read fasta records inside a file.

currecord=iterator.next()

The iterator working. The product of this instance is stored in currecord

secuencia=currecord.sequence

Sequence is a property of currecord, and we store it in the variable "secuencia".

arin.close()

Since the file is not longer needed, the filehandle is closed

a=seqab1[x][0]

b=seqab1[x][1]

To retrieve the first (and second) element in the list inside the dictionary. a and b are the start and end position of the vector in the sequence.

if len(seqab1[x])==2:

to check if the list in the dictionary has 2 elements, since there a list with 2 and 4 elements.

newseq=secuencia[:a]+secuencia[b:]

Here is the new sequence (with newseq variable name). It is composed from the original sequence, until the start of the vector, plus the end of the vector until the end of original sequence.

elif len(seqab1[x])==4:

To check if the list in the dictionary has 4 elements

c=seqab1[x][2]

d=seqab1[x][3]

newseq=secuencia[:a]+secuencia[b:c]+secuencia[d:]

As before, we build the new sequence, but removing the vector from both sides.

else:

pass

If there is a different number of start-end pairs, the program will do nothing. Since there is no biological meaning in such event.

arout=open('wovects/'+x[:-3]+'txt','w')

Next step is to write the new sequence, in fasta format, so this line opens a file for writing (in another directory). Nothing is writing up to this moment.

dna=Seq(newseq)

We define newswq as a sequence objet, with the name dna.

seq=SeqRecord(dna,id=currecord.title,description="")

SeqRecord will create a SeqRecord objet based on the seq object. SeqRecord allows to include more associated data, like ID, name, description, annotation and features.

sali=FASTA.FastaWriter(arout)

This will init the FASTA writer.

sali.write(seq)

This statement actually writes the sequence in FASTA format.

arout.close()

arin.close()

Closes all open files.

Melting Point Calculator

The Melting Point function will calculate the melting point of a DNA or RNA duplex, given the sequence and other mix parameters. It is part of Biopython. The main drawback is that you need to know biopython to use it. There are many ways to wrap up all the complexity of biopython functions for the end user. We will use a web approach since it is easy to setup and all users know how to use a web form. Just use any HTML or text editor to make the GUI. This form asks for the same parameters that Tm function uses. The data the user enters, will go to the Biopython program using CGI.

Form code:

In this slide we show the form portion of the HTML needed to input the data.

**<form method="post" action="cgitm.cgi">
**

This is the form declaration, in the action field we have to refer to where the actual python code resides.

Sequences: <textarea rows=10 cols=30 wrap=virtual name=seqs></textarea>

This text area is where we input the sequence(s). rows and cols defines the relative size in the screen. The name variable is very important since will be used later.

**Salt concentration: <input type="text" name="saltc" value="10" size="3">[mM]
**

**Nucleotide concentration: <input type="text" name="nucc" value="10" size="3">[nM]
**

Two new variables: saltc and nucc. The input type is text, and not textarea since is only a line of text.

Nucleotide type:

**<select name="nucle">
<option value="0" selected="selected">DNA</option>
<option value="1">RNA</option>
</select>**

This generate a dropdown box with two options (DNA and RNA) and assign a value for each instance (0 for DNA and 1 for RNA).

<input type="submit" name="SubmitButton" value="Calculate tm"></form>

This code will generate the submit button.

This is all the coded need to make the interface for the end user. Most of it can be done with any HTML editor (like Nvu).

Generate Tm in HTML from multiple sequences using Python

When the user click on the submit button, the following code is executed:

#!/usr/bin/python

This indicates where the python interpreter is located. This is not a standart comment. The server needs it to know how to execute the python code. Remember that CGI can execute any code (like BASH, Perl, C) so you need to explicitly define the path to the interpreter.

def Tm_staluc(s,dnac=50,saltc=50,rna=0):

This is the Tn function definition. Its displayed here to show the all the parameter it needs. The rest of the internal work of the function is not relevant for this exercise.

Explanation of cgi program.

formu=cgi.FieldStorage()

FieldStorage is a function (inside cgi module) that will store all the variables values from the html form.

**doc=SimpleDocument(title="Melting Temperature OUTPUT, **
Bgcolor=WHITE, cgi=1)

SimpleDocument is a function from the module Htmlgen, it defines an html page. The first parameter is the HTML title, the second is the background color and the last parameter, cgi, is useful if your page is for cgi use.

try:

The block under try is executed, but if any exception is raised, the except code is executed instead.

dseqs=formu["seqs"].value

This will retrieve the value of the variable seqs from the form.

fsaltc=float(formu["saltc"].value)

fnucc=float(formu["nucc"].value)

fdna=int(formu["nucle"].value)

As before, we are retrieving all the remaining values from the form.

except:

The following block of code will be executed only if there is any error during the execution of the "try" block.

```
dseqs="gtcttctgatctacatctgcgctatgc"
```

```
fsaltc=50
```

```
fnucc=50
```

```
fdna=0
```

Some default values, if the try block fails, all the variables will have a value.

```
doc.append('<pre>')
```

On this sentence we are appending an HTML element inside the webpage.

```
unvar=string.split(dseqs,"\\n")
```

unvar is now a list of input sequences. The '\\n' is the enter that separates each sequence.

```
for x in unvar:
```

To cycle inside all sequences

```
if len(x)>10:
```

Do this block only if the sequence is larger than 10.

```
theTM=Tm_staluc(x,fnucc,fsaltc,fdna)
```

This is when the Tm function is called, using the parameters entered using the form.

```
doc.append(str(theTM))
```

On the document, the Tm value (converted to string) is added.

```
else:
```

The following block (one line) is executed when the sequence (in x) is less than 10 nucleotides long.

```
doc.append("Sequence not long enough")
```

To add this warning when the sequence is small.

```
doc.append('</pre>')
```

After the list has been processed, the pre tag is closed.

```
print doc
```

This will print all the generated HTML code.

This is a screen capture of running the program from the command line. Since there is no form to retrieve the data from, the default values are being used. The first two lines that are printed, at first sight seems that there are not generated by our program. But they are pre-appended to the HTML output by the cgi=1 statement in SimpleDocument function.

Result of the CGI code output after pressing submit button is pressed in HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
```

```
<HTML>
```

```
<!-- This file generated using Python HTMLgen module. -->
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen 2.2.2">
    <TITLE>Melting Temperature OUTPUT</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
  <pre>
47.0052165167
48.8539424877
60.8652584543
56.6515506015
59.2044067674
  </pre>

</BODY> </HTML>
```


14th Annual International Conference On Intelligent Systems For Molecular Biology

ISMBS 2006 Fortaleza, Brazil
August 6-10, 2006
and 2nd Annual AB³C Conference: X-Meeting

Python programming for Life Science researchers

Sebastián Bassi,
UNQ, Argentina
sbassi@gmail.com

ISMBS 2006 - Fortaleza, Brazil - August 6-10, 2006

Why Python?

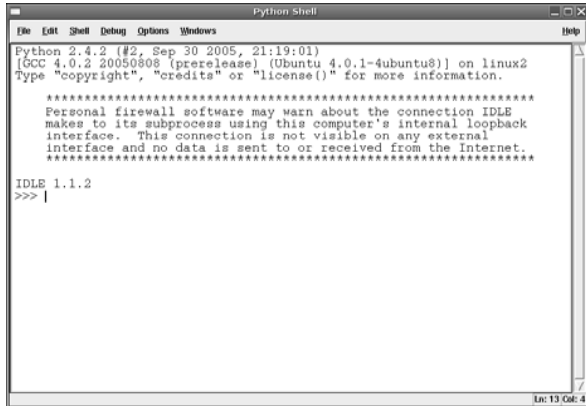
- Some python characteristics:
 - Easy to read (pseudocode that works)
 - Easy/Fast to write
 - Batteries included
 - Multiplatform
 - Dynamically typed
 - Strongly typed
 - Friendly community

ISMBS 2006 Fortaleza, Brazil
August 6-10, 2006

Python Overview: What is different from other languages

114

Python interactive interpreter



Python 2.4.2 (#2, Sep 30 2005, 21:19:01)
[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu8)] on linux2
Type "copyright", "credits" or "license()" for more information.

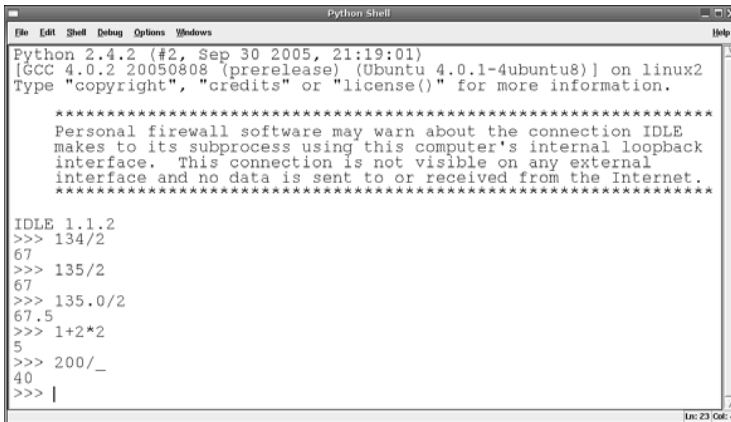
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.

IDLE 1.1.2
>>> |

Python interactive interpreter screenshot

115

Python as calculator



Python 2.4.2 (#2, Sep 30 2005, 21:19:01)
[GCC 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu8)] on linux2
Type "copyright", "credits" or "license()" for more information.

Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.


IDLE 1.1.2
>>> 134/2
67
>>> 135/2
67
>>> 135.0/2
67.5
>>> 1+2*2
5
>>> 200/_
40
>>> |

Python can be used as a calculator

116

Data types: Int, Long, Float

- Int: From $-2.10^{31}-1$ to $2.10^{31}-1$
- Long: Integer > than 2.10^{31}
- Float: Floating point numbers.



The screenshot shows a 'Python Shell' window with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help). The command prompt shows the following interactions:

```
>>> MyVariable=2147483647
>>> MyVariable
2147483647
>>> type(MyVariable)
<type 'int'>
>>> MyOtherVariable=2147483648
>>> type(MyOtherVariable)
<type 'long'>
>>> type(1.0)
<type 'float'>
>>> |
```

The status bar at the bottom right indicates '(In: 335 Out: 4)'.

Int, Long and Float numbers

Data Types: String

```
Python Shell
File Edit Shell Debug Options Windows Help

>>> FirstString="Hello World!"
>>> print FirstString
Hello World!
>>> print FirstString + " In Python"
Hello World! In Python
>>> print FirstString[0:5]
Hello
>>> print FirstString[:5]
Hello
>>> print "ISMB " + 2006

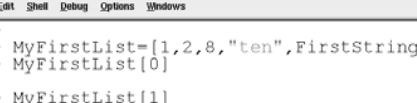
Traceback (most recent call last):
  File "<pyshell#207>", line 1, in <module>
    print "ISMB " + 2006
TypeError: cannot concatenate 'str' and 'int' objects
>>> print "ISMB " + str(2006)
ISMB 2006
>>> |
```

Strings can be concatenated like this:

```
'%S ... %S' % ('tga', 'atg')
```

Data types: Lists

- An array of data. Like C vectors, VB and Perl arrays.



The screenshot shows a Python Shell window with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help) and a command prompt. The following commands and their outputs are shown:

```
>>>
>>> MyFirstList=[1,2,8,"ten",FirstString]
>>> MyFirstList[0]
1
>>> MyFirstList[1]
2
>>> MyFirstList[2]
8
>>> MyFirstList[3]
'ten'
>>> MyFirstList[4]
'Hello world!'
>>> MyFirstList[3][0]
't'
>>> |
```

The status bar at the bottom right indicates "Ln: 81 Col: 1".

List, definition, creating and invoking

List: Notation

The image shows a Python Shell window with the title "Python Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area contains the following code:

```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> MyFirstList=[1,2,8,"ten",FirstString]
>>>
>>>         | | |         |         |
>>>         1 2 3         4         5
>>>
>>>
>>>
```

Below the main window, there is a smaller, overlapping window also titled "Python Shell". It shows the execution of slice operations on the list:

```
>>> MyFirstList[:4]
[1, 2, 8, 'ten']
>>> MyFirstList[-3:]
[8, 'ten', 'Hello World!']
>>>
```

The status bar at the bottom right of the overlapping window shows "Ln: 430 Col: 1".

Slice notation used for lists

List: Operations, insert data

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> MyFirstList
[1, 2, 8, 'ten', 'Hello World!']
>>> MyFirstList.append(99)
>>> MyFirstList
[1, 2, 8, 'ten', 'Hello World!', 99]
>>> MyFirstList.insert(2,40)
>>> MyFirstList
[1, 2, 40, 8, 'ten', 'Hello World!', 99]
>>> MyFirstList.extend([55,"last"])
>>> MyFirstList
[1, 2, 40, 8, 'ten', 'Hello World!', 99, 55, 'last']
>>> |
```

- Append: Add elements to the last position.
- Insert: Add in any arbitrary position.
- Extend: Add a list to the last position



Append, Insert, Extend as way to insert data in a list

121

List: Delete elements

```
Python Shell
File Edit Shell Debug Options Windows Help
>>>
>>> MyFirstList
[1, 2, 40, 8, 'ten', 'Hello World!', 99, 55, 'last']
>>> AInt=MyFirstList.pop(2)
>>> MyFirstList
[1, 2, 8, 'ten', 'Hello World!', 99, 55, 'last']
>>> AInt
40
>>> MyFirstList.remove("ten")
>>> MyFirstList
[1, 2, 8, 'Hello World!', 99, 55, 'last']
>>> |
```

- LIST.pop(n) will retrieve the n element of LIST (default=last)
- LIST.remove("N") will remove the first "N" in LIST



Delete with pop and remove. Pop will return the value, and pop() will do it with last element

122

Tuples

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ATuple=("gcacttc","cttcgc","caatgc")
>>> ATuple
('gcacttc', 'cttcgc', 'caatgc')
>>> ATuple[0]
'gcacttc'
>>> ATuple.pop()

Traceback (most recent call last):
  File "<pyshell#258>", line 1, in -tople
vel-
    ATuple.pop()
AttributeError: 'tuple' object has no attribute 'pop'
>>> |
```

- Defined like a list, with parentheses instead of square brackets.
- Indexes works as lists. Can use slicing.
- Tuples are immutable. Can't add or remove elements.
- Tuples are faster than list. Tuples are like "write-protected" list.



When you need to iterate over a list of constant values, use a tuple instead of a list.

123

Dictionaries

- Datatype to store one-to-one relationships between **keys** and **values** (like hash in Perl or the Scripting. Dictionary object in Visual Basic).

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> threecode = {'A':'Ala', 'B':'Asx', 'C':'Cys', 'D':'Asp',
'E':'Glu', 'F':'Phe', 'G':'Gly', 'H':'His',
'I':'Ile', 'K':'Lys', 'L':'Leu', 'M':'Met',
'N':'Asn', 'P':'Pro', 'Q':'Gln', 'R':'Arg',
'S':'Ser', 'T':'Thr', 'V':'Val', 'W':'Trp',
'Y':'Tyr', 'Z':'Glx', 'X':'Xaa', '*':'Ter',
'U':'Sel'}
>>> threecode["Q"]
'Gln'
>>> threecode.keys()
['*', 'A', 'C', 'B', 'E', 'D', 'G', 'F', 'I', 'V', 'H', 'K', 'M', 'L',
'N', 'Q', 'P', 'S', 'R', 'U', 'T', 'W', 'Y', 'X', 'Z']
>>> threecode.values()
['Ter', 'Ala', 'Cys', 'Asx', 'Glu', 'Asp', 'Gly', 'Phe', 'Ile',
'His', 'Lys', 'Met', 'Leu', 'Asn', 'Gln', 'Pro', 'Ser', 'Arg',
'Sel', 'Thr', 'Trp', 'Val', 'Tyr', 'Xaa', 'Glx']
```



threecode dictionary is part of Biopython. Elements in a dictionary are unordered.

124

Dictionaries: Some methods

- If key is not found, Python rises an error:

```
>>> threecode["kkk"]
```

Traceback (most recent call last):

```
File "<pyshell#299>", line 1, in <module>
    threecode["kkk"]
```

KeyError: 'kkk'

- Before looking for a value, check the key:

```
>>> threecode.has_key("kkk")
```

False



*del threecode["A"] deletes that item from dictionary.
threecode.clear() deletes all items.*

125

Program flow: If, elif, else

Code from Biopython

```
length = len(seq)
```

```
if length > 20:
```

```
    short = '%s ... %s' % (seq[:10], seq[-10:])
```

```
else:
```

```
    short = seq
```

```
if database in ['nucleotide']:
```

```
    format = 'gb'
```

```
elif database in ['protein']:
```

```
    format = 'gp'
```

```
else:
```

```
    raise ValueError("Unexpected database: %s" % database)
```



*See footnote on slide 6 for string concatenation and slide 8
for list slicing. Elif works as C switch*

126

Program flow: For

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> indent=4
>>> oinfo="ACTGGC"+"\\n"
>>> output_parts=["ACTTCG","GCTCTAG","TTCGAC"]
>>> for opart in output_parts:
>>>     oinfo += " " * indent + opart + "\\n"

>>> oinfo
'ACTGGC\\n    ACTTCG\\n    GCTCTAG\\n    TTCGAC\\n'
>>> |
```

for x in range(5) works as BASIC for x=0 to 4



*To cycle inside numbers, create a list with numbers with
range function. See indentation.*

127

While: Do while is true

```
while '' in new_tax_list:
    new_tax_list.remove('')
```

while True: will generate an infinite loop. Can be
escaped with break.



We will use "while True:" and break on BLAST parsers

128

Modularize your code: Functions

```
def get_interpro_entry( id ):
    """get specified interpro entry"""
    handle = urllib.urlopen("http://www.ebi.ac.uk/interpro/IEntry?ac=" + id )

    # XXX need to check to see if the entry exists!
    return handle
```

- Variables declared inside a function, lives only inside the function. Only argument in "return" is returned to the program.
- If the function just do something instead of returning a value use: return None (this is not mandatory, but improves legibility of the code)
- Usage: MyInterproHandle = get_interpro_entry("IPR004560")



To return more than one value, return a list with all the variables you need.

129

Modules

A chunk of code that can be used from a program or in interactive mode. Functions, classes, constants and dictionaries can be called and used from a program.

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> string.find("Hello, Python!", "Py")
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    string.find("Hello, Python!", "Py")
NameError: name 'string' is not defined
>>> import string
>>> string.find("Hello, Python!", "Py")
7
>>> |
```



Modules are searched in several path, like your home directory. See them all with sys.path.

130

Reading text files

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> MyHandle=open("/home/sbassi/wifi.log", "r")
>>> print MyHandle
<open file '/home/sbassi/wifi.log', mode 'r' at 0xb6e9d1d0>
>>> line=MyHandle.readline()
>>> line
'Mon Oct 24 22:49:24 2005 Entering network: cas
a25\n'
>>> MyHandle.close()
>>>
```

fileobject=open(filename,"r")
for line in fileobject:
 print line

readlines() return a list
of string from all the file



Files can't be edited while opened, until closed.

131

Write text files

```
Python Shell
File Edit Shell Debug Options Windows Help
>>> ANiceFile=open("mynewfile", "w")
>>> ANiceFile.write("This will make into the file\n")
>>> ANiceFile.close()
>>>
```

There are two modes for writing files:

- w: Write with overwrite if a file exists
- a: Write at the end of the file (append). Useful for logs.



Open can take a third argument, which defines how file is buffered before writing.

132

Data Manipulation

- The problem: A text file with data on it should be parsed, that is, read and interpreted by the program, and then display or store only selected information.
- Python tools:
 - Build-in open file function.
 - Control flow structures.
 - String manipulation methods.



This is a generic overview of the problem and tools.

133

Sample file: BLAST Hit table

```
inseq2  gjl26249933|ref|NP_755973.1| 100.00 29 0 0 1 29 837 865 1e-0860.8
inseq2  gjl1789736|gb|AAC76363.1| 100.00 29 0 0 1 29 834 862 1e-0860.8
inseq2  gjl3483131|gb|AAC33265.1| 100.00 29 0 0 1 29 480 508 1e-0860.8
inseq2  gjl29542596|gb|AAO91530.1| 46.4328 15 0 2 29 515 542 4.2 32.3
inseq2  gjl67762813|ref|ZP_00501511.1| 48.2829 15 0 1 29 278 306 7.2 31.6
inseq2  gjl67737420|ref|ZP_00488193.1| 43.1227 15 0 1 29 278 306 7.2 31.6
inseq2  gjl67714721|ref|ZP_00484082.1| 47.8842 15 0 1 29 278 306 7.2 31.6
inseq2  gjl69988727|ref|ZP_00641885.1| 41.3159 15 0 1 29 221 249 7.2 31.6
```

2000 more lines follows (removed to enter into this slide)

Your mission (should you choose to accept it): Get all GI from this file and retrieve URL to get full Genbank record only if “% identity” is greater than 45%.



This URL will be handy for this kind of task:
ncbi.nlm.nih.gov/entrez/query/static/linking.html

134

Python script of data manipulation

```
test3.py - /home/sbassi/notas/test3.py
File Edit Format Run Options Windows Help
import string
baseurl='http://www.ncbi.nlm.nih.gov/entrez/\
query.fcgi?cmd=text&db=protein&dopt=genpept&uid='
infile=open('allans.txt','r')
for line in infile:
    ids=string.split(line,'\t')[1]
    p_ident=float(string.split(line,'\t')[2])
    if p_ident>45:
        #retrieve only gi from ids
        gi=string.split(ids,'|')[1]
        print baseurl+str(gi)
    else:
        pass
Ln: 13, Col: 12
```



To send the output to a text file just redirect it in the command line with “>”.

135

XML: Basic Overview

- Language to describe data (with nothing about data presentation).
- Based on text format (binary XML is out of the scope of this tutorial).
- XML are “human-legible” (kind of)
- Easy to write programs to process XML documents
- Header with parsing information:
 - `<?xml version="1.0"?>`
- Body:
 - `<tagname attribute_name="attribute_value">a text</tagname>`
 - `<line type='demo'>A simple line</line>`
 - Empty element: ``



Pay attention: XML is everywhere!. Official webpage is
www.w3.org/XML

136

XML: Some real world samples

A RSS feed. Is XML based.

```
view-source: - Source of: http://rss.slashdot.org/Slashdot/slashdot - Mozilla Firefox
File Edit View
<item rdf:about="http://slashdot.org/article.pl?sid=06/05/01/213">
<title>'Revenge of the Nerds' Remake in the Works</title>
<link>http://rss.slashdot.org/Slashdot/slashdot?m=5222</link>
<description>grouchomarxist writes "According to CNN the movie '
&lt;p&gt;&lt;a href="http://rss.slashdot.org/~a/Slashdot/slashdot">
<dc:creator>CmdrTaco</dc:creator>
<dc:date>2006-05-01T22:26:00+00:00</dc:date>
<dc:subject>movies</dc:subject>
<slash:department>score-with-the-omega-mus</slash:department>
<slash:section>mainpage</slash:section>
```



RSS is a popular way to syndicate news. Atom is another protocol, also based on XML.

137

XML: Some real world samples

XML BLAST output.

```
view-source: - Source of: http://localhost/apache2-default/blast/blast.cgi - Mozilla Firefo
File Edit View
<?xml version="1.0"?>
<!DOCTYPE BlastOutput PUBLIC "-//NCBI//NCBI BlastOutput/EN" "NCB
<BlastOutput>
<BlastOutput_program>blastp</BlastOutput_program>
<BlastOutput_version>blastp 2.2.10 [Oct-19-2004]</BlastOutput_
<BlastOutput_reference>-Reference: Altschul, Stephen F., Thoma
<BlastOutput_db>nr</BlastOutput_db>
<BlastOutput_query-ID>lcl|QUERY</BlastOutput_query-ID>
<BlastOutput_query-def>inseq2</BlastOutput_query-def>
<BlastOutput_query-len>29</BlastOutput_query-len>
```



BLAST can be instructed to output as XML instead of text or HTML

138

XML: Sample with attributes

```
<svg xmlns="http://www.w3.org/2000/svg" width="800" height="600">
<circle cx="400" cy="200" r="150" style="fill:none;stroke:black;stroke-
width:0.5"/>
<text x="400" y="200" style="text-anchor:middle; font-size:12px; font-
weight:bold; font-family:Arial;">X6J44</text>
<text x="400" y="215" style="text-anchor:middle; font-size:10px; font-
family:Arial;">3455 bp</text>
<path d="M 511.62370399002 300.200542451368 L 530.227654655024
316.900632859929" style="fill:none;stroke:black;stroke-width:0.5" />
<text x="533.948444788024" y="320.240650941641" style="text-anchor:start;
font-size:9px; font-family:Arial;">ecoRI 1266</text>
```

All elements in this sample contains attributes. **SVG** contains **width** and **height**. Text contains **x**, **y** and **style** and Path has **d** and **style**.



Plasmids in SVG at: bioinformatics.org/savvy/. More bioXML at: xml.com/pub/rg/Bioinformatics

139

XML: Parser with elementtree

```
xmlparser.py - /home/vicky/xmlparser.py
File Edit Format Run Options Windows Help
from elementtree.ElementTree import ElementTree
myxml=open("slashdot.xml","r")
root=ElementTree(file=myxml)
iter=root.getiterator()
for ele in iter:
    if ele.keys():
        print ele.items()[0][0]
    else:
        print ele.tag
        print ele.text
```



Elementtree must be installed separately.

140

What is Biopython?

It is a distributed collaborative effort to develop Python libraries and applications which address the needs of current and future work in bioinformatics.

It provides:

- Tools for working with sequences (aa and nt).
- Parsers of all popular bio file formats (fasta, gb, pdb, BLAST output).
- Data retrieve from biological databases.
- Wrapper to bio-programs (BLAST, ClustalW, EMBOSS, Primer3, and more).
- Biological functions like LCC, restriction enzymes cutting, and more.
- Tables and constants.



With biopython you can program repetitive task concatenating several programs.

141

Biopython sample. BLAST output parsing for vector removing from DNA sequences

BLAST Search Results														
# BLASTN 2.2.10 [Oct-19-2004]														
# Query: pBlueSP														
# Database: Inseqsl														
# Fields: Query id, Subject id, % identity, alignment length, mismatches, gap openings, q. start, q. end, s. start, s. end, e-value, bit score														
# Query: pBlueSP														
pBlueSP	Q2_Martin_plate_B_76010.F_0660.abl	100.00	59	0	0	2204	2262	11	69	6e-27	117			
pBlueSP	Q1_Martin_plate_A_64408.F_0275.abl	98.39	62	1	0	2201	2262	7	68	2e-26	115			
pBlueSP	Q1_Martin_plate_A_58808.F_0275.abl	100.00	58	0	0	2205	2262	10	67	2e-26	115			
pBlueSP	Q2_Martin_plate_B_09402.F_0277.abl	98.44	64	0	1	2199	2262	4	66	4e-25	111			
pBlueSP	Q2_Martin_plate_B_68009.F_0660.abl	97.06	68	1	1	2195	2262	1	67	4e-25	111			
pBlueSP	Q2_Martin_plate_B_58807.F_0277.abl	100.00	55	0	0	2208	2262	12	66	2e-24	109			
pBlueSP	Q2_Martin_plate_B_21553.F_0277.abl	100.00	55	0	0	2208	2262	13	67	2e-24	109			
pBlueSP	Q2_Martin_plate_B_13E02.F_0277.abl	100.00	55	0	0	2208	2262	14	68	2e-24	109			
pBlueSP	Q1_Martin_plate_A_99412.F_0275.abl	100.00	55	0	0	2208	2262	11	65	2e-24	109			
pBlueSP	Q1_Martin_plate_A_93E12.F_0275.abl	100.00	55	0	0	2208	2262	11	65	2e-24	109			



Elementtree must be installed separately.

142

```
import String
from Bio import Fasta
from Bio.SeqIO import FASTA
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
arin=open("resultado2.txt","r")
seqabl={}
for line in arin:
    if ".abl" in line and "#" not in line:
        nomtemp=line.split("\t")[1]
        #print nomtemp
        if nomtemp not in seqabl.keys():
            seqabl[nomtemp]=[int(line.split("\t")[8]),int(line.split("\t")[9])]
        else:
            if seqabl[nomtemp]==[int(line.split("\t")[8]),int(line.split("\t")[9])]:
                print "repetido: "+nomtemp
            else:
                if int(line.split("\t")[9])>int(line.split("\t")[8]):
                    listemp=seqabl[nomtemp]
                    listemp.extend([int(line.split("\t")[8]),int(line.split("\t")[9])])
                    seqabl[nomtemp]=listemp
                else:
                    listemp=seqabl[nomtemp]
                    listemp.extend([int(line.split("\t")[9]),int(line.split("\t")[8])])
                    seqabl[nomtemp]=listemp
    else:
        pass
arin.close()
```



This first half parse the BLAST output, w/o using biopython.

143

```
for x in seqabl:
    arin=open("renames/"+x[:-3]+"txt","r")
    parser = Fasta.RecordParser()
    iterator=Fasta.Iterator(arin,parser)
    currecord=iterator.next()
    secuencia=currecord.sequence
    arin.close()
    a=seqabl[x][0]
    b=seqabl[x][1]
    if len(seqabl[x])==2:
        newseq=secuencia[a]+secuencia[b:]
    elif len(seqabl[x])==4:
        c=seqabl[x][2]
        d=seqabl[x][3]
        newseq=secuencia[a]+secuencia[b:c]+secuencia[d:]
    else:
        pass
    arout=open('wovects/'+x[:-3]+'txt','w')
    dna=Seq(newseq)
    seq=SeqRecord(dna,id=currecord.title,description="")
    sali=FASTA.FastaWriter(arout)
    sali.write(seq)
    arout.close()
arin.close()
```



Using fasta parser to read sequences and FastaWriter to write the modified sequence.

144

Melting point calc. by Sebastián Bassi

With HTML is easy to make GUIs to command line programs or Biopython functions. Just use any HTML or text editor. This form asks for the same parameters that Tm function uses.

```
catgctgtcatcatgcgcgatagcatcgat
cactagctagcatgcatcgatcgccacag
cgtacgagcattctatcatcgctacgat
cacggtctcgatctgaatgacaccactcag
cagctactgatgctgtgattc
ccggtatgcgatctgcatgcatg
```

Sequences:

Salt concentration: [mM]

Nucleotide concentration: [nM]

Nucleotide type:

The Tm function is inline to avoid dependency problem in some hosts.

145

Form code

```
<form method="post" action="cgi-bin/cgitm.cgi"><br>
Sequences: <textarea rows="10" cols="30" wrap="virtual" name="seqs"></textarea>
<br>
Salt concentration: <input type="text" name="saltc" value="10" size="3"> [mM]<br>
Nucleotide concentration: <input type="text" name="nucc" value="10" size="3"> [nM]
Nucleotide type:
<select name="nucle">
<option value="0" selected="selected">DNA</option>
<option value="1">RNA</option>
</select>
<br>
<br>
<br>
<input type="submit" name="SubmitButton" value="Calculate tm"></form>
```

Look for action path and variable names.

146

Generate Tm in HTML from multiple sequences using Python

```
#!/usr/bin/python
from HTMLgen import *
from HTMLcolors import *
import os
import cgi
import string
import math

def Tm_staluc(s,dnac=50,saltc=50,rna=0):
    """Returns DNA/DNA tm using nearest neighbor thermodynamics. dnac is
    DNA concentration [nM] and saltc is salt concentration [mM].
    rna=0 is for DNA/DNA (default), for RNA, rna should be 1.
    Sebastian Bassi <sbassi@genesdigitales.com>"""
    #Credits:
    #Main author: Sebastian Bassi <sbassi@genesdigitales.com>
    #Overcount function: Greg Singer <singergr@tcd.ie>
    #Based on the work of Nicolas Le Novere <lenov@ebi.ac.uk> Bioinform
    dh=0 #DeltaH. Enthalpy
    ds=0 #deltaS Entropy
```

The Tm function is inline since to avoid dependency problem in some hosts.

147

```
formu=cgi.FieldStorage()
doc= SimpleDocument(title="Melting Temperature OUTPUT", \
    bgcolor=WHITE, cgi=1)

try:
    dseqs=formu["seqs"].value
    fsaltc=float(formu["saltc"].value)
    fnucc=float(formu["nucc"].value)
    fdna=int(formu["nucle"].value)
except:
    dseqs = "gtctattgtgtatccgcgattcgcgcgatctaa"
    fsaltc = 50
    fnucc= 50
    fdna= 0
doc.append('<pre>')
unvar=string.split(dseqs,"\n")
for x in unvar:
    if len(x)>10:
        theTM=Tm_staluc(x,fnucc,fsaltc,fdna)
        doc.append(str(theTM))
    else:
        doc.append("Sequence not long enough")
doc.append('</pre>')
print doc
```

In formu are stored all the form variables. Doc is an object used for store the HTML info.

148

```
sbassi@vicky2:/var/www/apache2-default/cgi-bin $ ./cgitm.cgi
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<!-- This file generated using Python HTMLgen module. -->
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen 2.2.2">
  <TITLE>Melting Temperature OUTPUT</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<pre>
63.5885725727
</pre>

</BODY> </HTML>

sbassi@vicky2:/var/www/apache2-default/cgi-bin $
```



CGI output generated from command line. The CGI script should work under CLI

149

Result of CGI code after submit button is pressed in HTML.



CGI output generated from command line. The CGI script should work under CLI

150

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<!-- This file generated using Python HTMLgen module. -->
<HEAD>
  <META NAME="GENERATOR" CONTENT="HTMLgen 2.2.2">
  <TITLE>Melting Temperature OUTPUT</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<pre>
63.9172232903
64.9429203546
59.534719514
69.8722552748
52.3744976098
58.8079358913
</pre>

</BODY> </HTML>
```



CGI output generated from command line. The CGI script should work under CLI

151

That's all for today. But there is a lot more in Python!

Resources:

The Quick Python Book, Dary Harms and Kenneth McDonald, Manning, 2000
 Professional XML, Birdbeck et al., 2nd Ed., Word Press, 2001
 Python Tutorial, Guido van Rossum, March 2006 (<http://docs.python.org/tut/>)
 Dive into Python (diveintopython.com)
 Biopython tutorial and cookbook, Jeff Chang, Brad Chapman, Iddo Friedberg, 2001 (<http://bioweb.pasteur.fr/docs/doc-gensoft/biopython/Doc/Tutorial.pdf>)
 Python Speed & Performance Tips (<http://wiki.python.org/moin/PythonSpeed/PerformanceTips>)
 Python course in Bioinformatics, Katja Schuerer, 2004 (<http://www.pasteur.fr/recherche/unites/sis/formation/python/>)



The Tm function is inline to avoid dependency problem in some hosts.

152