Tutorial on Creating Concrete ILP Formulations using Python

for the book:

Integer Linear Programming in Computational and Systems Biology

by Dan Gusfield

Published by Cambridge University Press, 2019

1 Introduction

This short tutorial on Python accompanies the book: *Integer Linear Programming in Computational and Systems Biology: An Entry-Level Text and Course* by Dan Gusfield, published by Cambridge University Press, 2019.

The book is also accompanied by about fifty computer programs written in Python and Perl, that can be *used* without knowing anything about computer programming or the languages Python and Perl. However, for the reader who *does* want to do more than use the programs, I have written a short tutorial on the *subset* of Python needed to understand two of the Python programs used in the book. It is not a complete tutorial on Python, but even discussing just these two Python programs exposes a substantial amount about procedural computer programming and the Python language.

The two Python programs that we discuss are *CLgraphfile.py* and *firstRNA.py*, which can be downloaded from:

www.cambridge.org/9781108421768

The awful thirteen digit number at the end is the ISBN number for the book.

The first program concerns the problems of finding a maximum-size clique in a graph (a task that is common in the analysis of biological, and social, networks); and the second program concerns the problem of finding a most stable folding of an RNA molecule.

2 Task C

In the Preface of the book, I enumerated four tasks, *A* to *D*. The book focuses on tasks *A, B* and *D*. Now I briefly discuss Task *C*, the task of writing a computer program that takes in a concrete description of a problem *instance*, and creates the *concrete* ILP formulation that can be input to an ILP solver.

If you are an experienced computer programmer (even if you do not know Python), you will already have a good idea of what such a computer program would look like and how it would work. But, readers who have no experience in computer programming probably have little sense of how a computer program takes in a concrete problem instance, and produces the required concrete ILP formulation. This tutorial is mostly written for those readers. It assumes no prior background in computer programming.

Goals The tutorial has two goals. First, to complete the presentation of the full *workflow* involved in the ILP approach to solving biological problems, answering the question of how concrete ILP formulations are efficiently created.[1] Second, to teach just enough Python programming that readers can start writing simple Python programs to produce concrete ILP formulations, and learn enough about Python and computer programming that they can learn more by themselves.

Why Python The choice of Python is somewhat arbitrary. It is a language that is now frequently use in bioinformatics, although its immediate ancestor, Perl, was until recently the main programming language for bioinformatics[2] (and *C* is sometimes unavoidable). Python style can seem a bit weird, but most of the concepts in Python programming are common to many modern programming languages, and it is a good place to start learning computer programming.[3]

In writing the Python programs for the book, I tried use the simplest Python possible. But, those programs cover a fair amount of Python, and

[1] writing them out manually is impossible for all but the smallest problem instances.

[2] I had planned to use Perl for this book, but by the time I got started, it was clear that Python had largely displaced Perl.

[3] We use the variant of Python numbered 2.71. This is a widely used version, but Python 3 is also widely used. For our purposes, the differences between the versions are very modest, but unfortunately programs written for Python 2.7 sometime will not run correctly under Python 3, and vice verse.

introduce almost all of the style of writing *procedural* programs.[4]

3 A First Exposure to Computer Programming

Here we begin a discussion of the computer programming language *Python*. We will focus on two problems: the *maximum-clique problem* discussed in Chapter 2 of the book; and the simple RNA folding problem, discussed in Chapter 6. In the case of the maximum-clique problem, a concrete problem instance is specified by a binary matrix describing the adjacencies (edges) in an undirected graph. and in the case of the RNA folding problem, the input is simply an RNA sequence.

We first discuss the Python program *CLgraphfile* that takes in a binary matrix of 0s and 1s describing an undirected graph G, and produces a concrete ILP formulation for the *maximum-clique* problem on graph G. The maximum-clique problem was discussed in the book in Chapter 2. I will assume that you have read and understood Section 2.2 of the book.

Program CLgraphfile.py My approach to teaching computer programming is to write and explain a few lines of code at a time, and let the reader induct from those examples and comments, rather than explicitly and formally detailing all the rules and variations, as one finds in a manual (or in many texts on computer programming). With examples and experimentation, letting the student use inductive learning, with a peek at a manual when needed, a motivated student learns faster this way.

To start, look at the program *CLgraphfile.py* in Figure 1. It takes in a file containing a matrix representing a graph G, and produces a concrete ILP formulation for the maximum-clique problem on graph G.

While learning about *CLgraphfile*, it is helpful to download the program from the book webpage and execute the program on small data, and then examine the concrete ILP formulation that is produced.

Execution Recall that *CLgraphfile.py* is executed by issuing the following command on a command line in a terminal window:

[4]You probably don't know what is meant by "procedural programs", and you don't need to. At this point, it only means that we won't be writing *object oriented* programs, which is a style of programming I abhor, and that you may have heard about (much more favorably).

```
# program CLgraphfile

import sys

INFILE = open(sys.argv[1], "r")
OUTFILE = open(sys.argv[2], "w")

constraints = "such that \n\n"
listC = ""
binaries = "binary \n"
index = 0

for line in INFILE:
    index = index + 1
    listC = listC + "+ C(%d) " % index
    binaries = binaries + "C(%d)\n " % index

    j = 0
    for char in line:
       j = j + 1
       if char == '0' and (index < j):
          string = "C(%d) + C(%d) <= 1\n" % (index,j)
          constraints = constraints + string

OUTFILE.write("Maximize \n")
OUTFILE.write(listC + "\n")
OUTFILE.write (constraints)
OUTFILE.write (binaries)
OUTFILE.write ("end")

print ("The ILP file is: %s \n" % sys.argv[2])

INFILE.close()
OUTFILE.close()
```

Figure 1: The Python program *CLgraphfile.py* that creates concrete ILP formulations for the maximum clique problem. Program *CLgraphfile.py* can be downloaded from the book's webpage. That version has comments that have been removed here. Removing those comments makes the Python stand out more, but the comments may be helpful, so you should also look at that version of the program.

```
python CLgraphfile.py adjacency-matrix-file ILP-file.lp
```

where "adjacency-matrix-file" is the (user-provided) name of a file that holds the adjacency matrix description of the input graph G, and "ILP-file.lp" is the (user-provided) name of the file that will hold the concrete ILP formulation. That file name must have the ".lp" extension. Those two names are called *arguments* of (or for) program *CLgraphfile.py*. Thus we say that *CLgraphfile.py* is a program with two arguments.

Before issuing the terminal command, be sure that first you have changed directories (for example with "cd" commands on a Mac) to where the program "CLgraphfile.py" resides. Otherwise, the system might not know where the program is.[5]

When the program is run, the statements in the program are executed from the *top* to the *bottom* of the program, except as modified by *iteration* and *conditional* statements, which alter the basic top-to-bottom "flow of control". This will be explained in detail below. But, first we have to explain even more basic elements of the program.

3.1 Variables, Values and Assignments

The most basic element of any programming language is the *variable*, which (for our purposes) is just a *name* that can be *assigned* to hold a value (essentially like a variable in algebra or an ILP formulation).

In Python, there are several different *types* of values that a variable can hold. We will only use two types in our first program. Those are the *integer* type, and the *string* type. An integer type is just an integer number (duh!), such as: 17; and a string type is an alpha-numeric string, such as: "this is a string" (without the quotation marks).

Since a variable can hold a value, we need a way to *assign* a value to it. For a variable named *index* , the statement:

```
index  = 0
```

has the effect of *assigning* the value of zero to the variable named *index*. For a variable named *row*, the statement (with the quote marks explicitly part of the statement):

```
row = "000100010"
```

[5]There are other ways to be sure that the system finds the program, but this is the simplest.

has the effect of assigning the string "000100010" (without the quote marks) to the variable *row*.

Note that the assignment statement is asymmetric - it takes a value from the *right* side of the equality sign, and assigns that value to the variable on the *left* side of the equality sign. The reverse is never correct. Thus

```
"000100010" = row
```

would give an error. Try it and learn.

OK, so we can assign values to a variable. How do we use those values? The simplest use is to print out the value a variable is holding. The following statements:

```
index = 17
print index
```

will print the number 17 on the computer terminal just below the print statement. Write the above two lines into a file, say "test.py", and then in a terminal window type the command:

```
python test.py
```

to see the effect. Note that the file name "test.py" has extension ".py". That extension is necessary for any file that holds a python program. Without that extension, the python system might not be willing or able to execute the program.

Nothing to Declare In Python, you do not have to *declare* whether a variable is intended to hold an integer or a string (as you must do in many other computer programming languages). Python figures out itself what type of data is being held. Moreover, the type is allowed to change. For example, it is fine to write:

```
j = "Now a string"
print j
j = 17
print j
```

Try it out.

As you may notice in looking at the print statements in *CLgraphfile.py*, they are more complicated than the simple print statement used here. We will explain all of that in due course. But one piece of the complication is the use of the two-character symbol "\n", called a "new line" symbol. It has an effect when something is printed. Its effect is to insert a blank line into what is printed. For example, if we change the four lines above to:

```
j = "Now a string \n\n"
print j
j = 17
print j
```

then what will be printed is:

```
Now is a string


17
```

Compare this to the previous output, when the new line symbols were not used.

Assignments in CLgraphfile.py Having learned a bit about assignment statements, let's look at the following four statements that in *CLgraphfile.py*.

```
constraints = "such that \n\n"
listC = ""
binaries = "binary \n"
index = 0
```

In these statements, the terms "constraints", "listC", "binaries" and "index" are names of (user-specified) variables. The first statement assigns the value "such that \n\n" to the variable named "constraints"; the second statement assigns the empty string to the variable "listC" (we will later add to this string, so that it will not always be empty); the third statement assigns the string "binary \n" to the variable "binaries"; and the fourth statement assigns the value 0 to the variable "index".

Magical incantations Before those four assignments, there are two more complex looking assignment statements:

```
INFILE = open(sys.argv[1], "r")
OUTFILE = open(sys.argv[2], "w")
```

The strings "INFILE" and "OUTFILE" look like variable names, and these two statements seem to be assigning something to them. But what? Remember that *CLgraphfile.py* is called (on a command line) with two arguments, which are both names of files. The effect of the statement:

```
INFILE = open(sys.argv[1], "r")
```

is to assign the first argument, which is a string, to the variable "INFILE". Moreover, it opens the file named in the first argument so that its contents can be read. Similarly, the effect of the second statement assigns the second argument to variable "OUTFILE", and opens the file named in the second argument so that strings can be written into it.

For example, the if I call program *CLgraphfile* (which is in the file "CLgraphfile.py")with the command

```
python CLgraphfile.py graph-matrix CLgraph.lp
```

then when the program executes the two assignment statements above, the variable "INFILE" will be assigned the string "graph-matrix", and variable "OUTFILE" will be assigned the string "CLgraph.lp". Further, the file named "graph-matrix" will be opened so that its contents can be read; and the file named "CLgraph.lp" will be opened so that strings can be written into it.[6]

Note the use of the "w" and "r" in the statements, telling the program whether a file will be read (r) from, or written (w) to. How we actually read and write will be discussed shortly.

One point to emphasize, in these two assignment statements, you have to literally use the terms "sys.argv[1]" and "sys.argv[2]" to refer to the two arguments used when *CLgraphfile* is executed.[7] But, the name of the variable on the left side of the statement is any variable name specified by the user.[8]

[6]If the input file "graph-matrix" does not exist, or is not in the same directory where the Python command is given, then the system will report an error. However, if the output file "CLgraph.lp" does not exist, the system will create it in the directory where the Python command is given.

[7]If the program had more arguments, they would be referred to as "sys.argv[3]", etc.

[8]I like to use the word "INFILE" for input files, and use the word "OUTFILE" for files used for output. But this that is just to remind me of their purposes. I could

It's magic I don't try to understand how this part of Python works. I just consider statements involving "sys.argv" to be magic incantations that assign the argument strings (given on a command line) to variables, so that the variables can be used later in the program. Also, this magic requires including the first statement in *CLgraphfile.py*, which is "import sys". That statement causes some software written elsewhere to be included in *CLgaphfile.py*. That software is necessary for the *sys.argv* command to work.

Note about specifying files to be opened The following statement will be used in an upcoming example:

```
INFILE = open ("example-file", "r")
```

In this "open" statement, the name of the file, "example-file", is literally written in the statement. So, in this example, the file to be opened is not specified on a command line. The effect is still to open file "example-file" to be read, and to assign string "example-file" to the variable "INFILE". Similarly, files that will be written to can be literally named in an open statement.

End of magic (for now) So much for magic incantations – just learn them and use them as needed. Now we can talk about concepts and syntactic details that make some intuitive sense.

Incrementing a value Program CLgraphfile contains the statement:

```
index = index + 1
```

What this statement does is *increase* by one the value being held in variable *index*. More completely, the current value of the variable *index* is added to 1, and the result is assigned to variable *index*. The effect is to *increment* the value held in *index* by one.

It may seem strange to have the variable *index* on both the left and right side of the equality sign, and certainly the specific statement above would make no sense in ordinary arithmetic. But, such statements are ubiquitous in computer programming, and illustrate a major way that Python (and most other computer programming languages) differ from natural languages. This statement is not asserting an equality – it is specifying the following *operations*:

---

just as easily used "BEATLEJUICE" in place of "INFILE". Also, it is common, but not necessary, to use all upper case letters for variable names used in input or output functions.

> Take the current value held in the variable *index*, add one to it, and assign the result back to the variable *index*.

The format of these kind of operational statements is that the *left* side contains only the name (or names, as we will see later) of variables, while the *right* side contains the operation(s) that create the value(s) to be assigned to the variable(s) on the left side.

4 Iteration and Blocks

The next construct of Python that we want to explain is the construct of *blocks*. But in order to do that we must also explain the concept of *iteration*.

4.1 Iteration

One of the most fundamental concepts, and heavily used construct, of any computer programming language is that of *iteration*, i.e., the *successive* repeating of some statement(s) in the program. Consider the following program called *testpy1*, which is in file *test.py*. It prints out each line in a file called "example-file", and also prints out a number indicating which line in the file was printed.

```
# program testpy1
#
import sys
INFILE = open ("example-file", "r")
i = 1
for line in (INFILE):
    print line
    print i
    i = i + 1
```

If file "example-file" contains the following three lines:

```
This is line 1
That was line 1
The previous line was line 2
```

then the resulting output of the program will be:

```
This is line 1

1
That was line 1

2
The previous line was line 2

3
```

How does program *testpy1* work? Remember that it is in a file that we have called "test.py". From our earlier discussion, we know that the first statement opens the file "example-file" for reading, and assigns the string "example-file" to variable "INFILE". The second statement assigns the value of 1 to the variable "i". Next comes the "for" statement with some syntax that we have not seen before.

In the statement:

```
for line in (INFILE):
```

the words "line" and "INFILE" are variables. This "for" statement has the general form of:

```
for variable1 in (variable2):
```

where *variable2* holds the name of a file that is open for reading. The effect is that each successive line in the file will be read in and assigned, as a string, to the variable "line".[9]

However, *between* each time that a line is read in from the file, the three statements below the "for" statement will be executed. Those three statements form what is called a *block*. You should be able to understand the actions of the statements in the block from our earlier discussions of print and iteration statements.

To repeat, the program first reads line 1 in the file, and then executes the statement in the following block; then it reads in line 2 and executes the block; and then it reads in line 3 and executes the block; and then the execution of the "for" statement is finished because there are no more lines in the file. Be sure that you see that these actions will produce the output shown above.

4.2 What exactly is a block?

When the above "for" statement is executed, the program *repeats* (iterates) the three statements that follow the "for" statement. Those three statements form a *block*. Every iteration (controlled by the "for" statement) executes *all* three of the statements in the block. But how does Python know which statements are in the block controlled by (and attached to) the "for" statement?

In Python, the statement before the beginning of a block ends with a colon (":"). The colon signals that a new block is about to start. Also, the position of every statement in the block is *indented* to the right, relative to the starting position of the statement before the block. We see that in program *testpy1* above.

For now, we will assume that there is only one block. In that case, every statement in the block must be indented the *exact same* number of spaces, and we see this in *testpy1*. If you do not obey this rule, you will get an error message when you try to run Python. For example, if you try to execute the following:

[9] I like to use variable names, such as "line", that suggest to me how the variable is being used. But the actual variable name, "line", has no meaning to Python. If I had used the variable name "blat", the "for" statement would still successively read in a line from the file and assign it to *blat*.

```
#program testpy1
#
import sys
INFILE = open ("example-file", "r")
i = 1
for line in (INFILE):
    print line
  print i
    i = i + 1
```

then you will get the following response:

```
 File "test.py", line 5
    print i
          ^
IndentationError: unindent does not match any outer indentation level
```

If instead, you indented line 5 to the right of the start of line 4, then you would get the error message:

```
File "test.py", line 5
    print i
    ^
IndentationError: unexpected indent
```

So, assuming there is only one block, as in the above example, *every* statement in the block *must* be indented exactly the same number of spaces. Now, I have to insert a word of caution here – Python has what I consider to be an awful design flaw (but I suspect some people think it is a feature, not a bug). The caution boils down to:

```
NEVER use the tab key to create spaces.
ONLY use the space bar to create spaces.
```

Using a mixture of tabs and spaces will work on the computer system where your write your program, but when you try to run your program on a different system, Python might declare that the program has an error of indentations that do not match. That is because a single *tab* is equivalent to a certain number of spaces, and different systems sometimes translate a tab to different number of spaces. So, if you want your Python program to work on all systems, avoid this problem by *never* using a tab.[10]

[10]Python purists will disagree and offer various fixes, but I think it is best to just avoid the problem.

How does the "for" statement work? OK, so we now explain the "for" statement used in the example, to explain how many times the block is executed.[11] In the statement:

```
for line in INFILE:
```

the word "line" is a variable name, selected by the programmer.[12]

Similarly, "INFILE" is a variable that holds the name of the input file, i.e., "example-file". The input file consists of an ordered list of lines of text. So, the effect of the statement "for line in INFILE" is that each successive line in *example-file* will be read, and assigned to the variable named "line"; and the block attached to the "for" statement will be executed each time a new line from the file is read. Finally, when all the lines in *example-file* have been read, this execution of the "for" statement, and the execution of the block attached to it, ends. Note that the words "for" and "in" are part of the Python syntax, while the words "line" and "INFILE" are variable names chosen by the programmer.

Two other examples of blocks Before returning to *CLgraphfile.py*, I want to describe another example of *iteration* and a block; followed by an example of a *conditional statement* and a block. Consider the following example:

```
i = 1
for char in line:
    print char
    print i
    i = i + 1
```

This looks very similar to the earlier statements:

[11] Wait, didn't we explain this already, a few paragraphs before? Yes. Forgetting that I had already written one, I wrote a second paragraph to explain the "for" statement. Then, I was going to delete one of them, but I've heard that repetition is the key to learning, hence I've left both in. So, if you thought you must not have understood something in the first explanation because you are now seeing a second explanation, this is the explanation of the explanations.

[12] The variable name "line" is a good choice because it is suggestive of what "line" is assigned. This helps us humans, reading or writing the program, to understand the logic of the program. But the word "line" has no meaning to Python – if we substitute (everywhere) the word "tablecloth" for "line", the program will run just fine and produce exactly the same output.

```
i = 1
for line in INFILE:
    print line
    print i
    i = i + 1
```

But in the new statements, the variable "char" has replaced "line", and "line" has replaced "INFILE". The replacement of "line" with "char" looks uneventful, but the replacement of "INFILE" with "line" looks like it could have a larger impact. The reason is that "line" holds a string, while it has been established (in the "open" statement) that "INFILE" holds the string which is the name of a file. So Python knows to interpret the two "for" statements differently.

What happens in each iteration of the new statements, is that the next unread single *character* of the string held in variable *line* is assigned to the variable *char*, and then the block attached to the "for" statement is executed. So, if the string held in variable *line* is:

```
This is line 1
```

then what will be printed is:

```
T
1
h
2
i
3
s
4

5
i
6
s
7

8
l
9
i
```

```
10
n
11
e
12

13
1
14
```

Notice that the character in positions 5, 8, and 13 is a space. However, you may notice that no other spaces are output, while in the first example of a "for" statement, there were added lines. That difference will be explained later.

4.3 Conditionals and blocks

The next example of a block is with the use of a *conditional* statement. Consider the following:

```
if char == "0":
   print "The character is 0, but we will change it to 1"
   char = "1"
   print char
```

The first statement starts with the word "if" and ends with a colon. The three following statements are indented and form a block. The effect of this is:

> *If* the value held in variable *char* is the *string* "0", *then* execute all of the statements in the block. If the value in *char* is not the string "0", then do *not* execute the statements in the block – just move past them.

The "if" statement and its relationship with the block attached to it is almost self-explanatory, with two important points to note.

First, the equality symbol is "==", i.e., two equal signs, not just one. The distinction is that a *single* equal sign is used to *assign* a value from the right side of the equality to a variable on the left side, as we saw earlier. The *double* equal sign is used to *test* whether the current value held in the variable on the left side of the double equality is the same as the value

on the right. Many computer languages use this convention to distinguish between assignment and testing. Confusing those uses is one of the most common errors in computer programming, particularly using "=" when "==" should be used.[13]

Second, the zero in the "if" statement is in *quotes*. That tells Python to test if the current value in *char* is a *string* consisting of a single character, "0", rather than the *integer* 0. If instead the "if" statement had been:

```
if char == 0:
```

then the effect would be to test whether the current value of "char" is the *number* zero. To emphasize the distinction consider the following:

```
char = 1
if char == 1:
   char = char + 3
   print char
```

This little program will print the number 4. But if we change the "if" statement to

```
if char == "1":
```

The "if" statement will evaluate to *false* (the test will fail), and so the block will not be executed and nothing will be printed. Try it out.

Conditional statements Generally, statements such as the "if" statement used above are called *conditionals*. When a conditional evaluates to *true* (passes the test), then the attached block is executed; and when a conditional evaluates to *false* (fails the test), then the attached block is skipped.

4.4 Blocks Inside Blocks

Above, we assumed that there was just one single block associated with a "for" or "if" statement. That was to simplify your first exposure to blocks. But in fact, a block may contain within it, another block, which itself may contain another block, etc. Such *nested* blocks are extremely common and useful. To illustrate this, let's consider the following problem:

[13] Using "==" when "=" is intended is generally not as bad, because the statement will usually not be grammatical and so the system will flag it as an error.

> We are given a file containing multiple lines of text. We need to read in each line of text, and for each line, print out the line, and its line number in the file; and then print out the characters in the line, numbering each one, as before. This is a combination of the two problems considered above.

A Python program to solve the stated problem could use two "for" statement and two *nested* blocks:

```
# program testpy2
import sys
INFILE = open ("example-file", "r")
i = 1
for line in INFILE:
    print line
    print i
    i = i + 1
    j = 1
    for char in line:
        print char
        print j
        j = j + 1
```

Note the statements from "print line" to "for char in line" are all indented, with the same indentation. They are all in the same block, attached to the first "for" statement. But the lines after the second "for" statement have a greater indentation. Why? Those lines form a *second* block, one that is contained *inside* the first block. The second block contains the three lines following the second "for" statement, which ends with a colon (marking the beginning of a block to follow). Then, the next three lines are all indented the same amount, which is larger than the indentation of the second "for" statement.

So, it is perfectly fine, and typically very useful, to have one block nested inside of another block. We say that the first block contains *all* of the statements from "print line" to "j = j + 1", and that the first block contains the second block.

Now let's look more closely at how this program fragment is executed. Suppose that the file specified by the variable "INFILE" contains two lines, say:

```
abcd
efgh
```

The execution first assigns variable "i" the value 1. It then proceeds to the next line, which is the first "for" statement, where variable *line* is assigned the first string "abcd" in the file whose name is the value of the variable "INFILE". Next, the program prints the string held in "line" (which is "abcd"), and then prints the value of variable i (which is 1). It then increments variable i to have value 2, and sets the variable j to value 1.

To recap, the program has encountered and executed statements in a top-down manner, and at this point the values of the variables are:

```
i:  2
line:  "abcd"
j:  1
```

Continuing with our examination of the program, remember that the execution is still inside the first block. But now we encounter a second "for" statement, and a new block that is attached to it:

```
    for char in line:
        print char
        print j
        j = j + 1
```

The effect of this code fragment is to successively print each character in the string held in variable *line*, where each character is followed by an integer giving the position of the character in the string. So, this prints:

```
a
1
b
2
c
3
d
4
```

When that printing is done, execution of the "for char in line" statement is done – for now. This is where things get interesting. At this point, the program has finished the execution of the second "for" statement, but the execution is still inside the first block, attached to the "for line in INFILE:"

statement. In fact, the end of the second block is also where the first block ends. So, control of the program returns to the first "for" statement, which continues where it last left off.

In more detail, the first line of the input file has been read, but there is another unread line in INFILE. So, the first "for" statement is not finished. Continuing where it left off, the second line in INFILE is now read and assigned to the variable *line*. The program next prints the second line, followed by the number 2. It then increments variable i, making its value 3. The next statement in the program is "j = 1", so the value of j *changes back* from 5 to 1.

At that point, the program encounters the "for char in line" statement that it previously encountered and executed. What happens? What happens is that this "for" statement is executed *again*, independent of its prior execution. But now, variable *line* has a different value ("efgh") than before. The result is that the following is printed:

```
e
1
f
2
g
3
h
4
```

Now control returns to the first "for" statement. But since all the lines in INFILE have been read, the first "for" statement is finished, and everything in the block attached to it is skipped. Skipping over a block means that the next statement to be executed should be the first statement after the end of that block. In the case of this program, there is no statement after the end of the first block, so the program is at its end, and the execution terminates.

4.5 Three nested blocks

OK, so we have seen two nested blocks. Now let's go for three. Consider the following code fragment:

```
# program testpy3
import sys
INFILE = open ("example-file", "r")
counta = 0
i = 1
for line in INFILE:
    print line
    print i
    i = i + 1
    j = 1
    for char in line:
        print char
        print j
        j = j + 1
        if char == "a":
           counta = counta + 1
print "The number of times that character ‘a’ appears in the input file is"
print counta
```

When program *testpy3* is executed, it will print out the lines we have discussed above, and then will print out the two additional lines:

```
The number of times that character 'a' appears in the input file is
3
```

Note that only *single* quote marks are used around the "a" inside of the print statement. This is because the double quote marks are needed at the start and end of the text that is to be printed. If "a" is enclosed in double quotes, Python would be confused and not produce what we want. You should cut and paste program *testpy3* into a file, called say *testpy.py*, and execute it, and modify it, to see for yourself what happens.

4.6 More refined output

The next thing I want to introduce is one way to produce less clunky output in Python. In program *testpy3* (and the earlier programs) each

print statement caused the value of a single variable to be printed, and on its own line. The result was readable, but not desirable. Suppose that instead of what is produced in *testpy3*, we want to print

```
There is an occurrence of character 'a' on line 3 in position 3 of that line
There is an occurrence of character 'a' on line 3 in position 7 of that line
There is an occurrence of character 'a' on line 4 in position 20 of that line
Character 'a' appears in the input file 3 times
```

The key thing to note is that each printed number is inside of a complete sentence, with words around it. This makes for nicer output of the kind that is essential in program *CLgraphfile*. To produce such output, we need to learn a little about how Python specifies *formatted output*. The program that produced the above output is *testpy4*, shown here:

```
# testpy4

import sys
INFILE = open ("example-file", "r")
counta = 0
i = 1
for line in INFILE:
    i = i + 1
    j = 1
    for char in line:
        if (char == "a"):
           print "There is an occurrence of character 'a' on line %d in position %
           counta = counta + 1
        j = j + 1
print "Character 'a' appears in the input file %d times" % counta
```

Let's look first at the second print statement, at the end of the program. Compared to the print statements we have seen so far, there are several new elements. First, there is the "%d" (percentage symbol followed by "d") between the words "file" and "times", and the statement ends with "% counta". Well, "counta" is a variable name, but what are the other things?

The symbol combination "%d" is a kind of *place holder*, telling Python that when the print statement is executed, and a line is printed, the value of some variable will go there. So, the symbols "%d" are not printed, but are replaced by the value of a variable. OK, which variable? You probably

can guess from the statement. After the last quotation mark, the symbol "%" indicates that the needed variable is listed next. In this case, the variable is "counta", and you can see that its value has been printed in the output above. Being able to print the value(s) of variables *inside* a longer text string allows for much nicer output, and is actually required to produce the correct output in program *CLgrahfile*.

There is one technical point we need to mention here. The two symbols "%d" do not just indicate that the value of a variable will be inserted there, they specify that the variable will have a value that is a *number*. In case that the value is a string, we have to use a different placeholder: "%s". An example of this is in the last print statement in program *CLgraphfile* shown in Figure 1.

Now let's look at the first print statement in *testpy4*. It has two occurrences of "%d", so when that line is printed, there will be two values of variables that will be inserted into the line. How are those variables specified? As you can guess, the two variables are listed, in the order that their values should appear in the line, after the single "%". Those variables are i and j. In the program, variable i is used to specify a line number in a file, and variable j is used to specify a position in a line.

Note that the variables are listed with a comma between them, and are enclosed in left and right parentheses. This is the standard way that Python specifies a kind of sequence called a *tuple* of things (in this case, a tuple of variables). I think of a tuple as a kind of list, but formally, in Python-ese, it is a sequence. We will say a bit about Python lists later. You should experiment with the program - removing the parentheses for example, to see what happens. The second print statement did not need parentheses because there was only one variable in the "tuple". Could it have been enclosed in parentheses also? Try it.

4.7 Two logical points

I want to point out two other details in program *testpy4*, which illustrate *logical* aspects of the program, rather than *syntactic* details. What I mean by *logical* is that if we get those details wrong, Python might not give an error message – because the program will obey all of the syntactic (grammatical(rules of Python – but the program might *not* produce the result we want. The first detail I want to point out is that the statement:

```
j = j + 1
```

is *not* part of the third block, but it is part of the second (and hence also, the first) block. We see that it is not part of the third block because it is not indented as far as the statements that are in the third block. It is indented the same way that the statements in the second block are indented.

If, by (logical) mistake, we had indented the line so that it was part of the third block, Python would not object – that would by syntactically ok - but when executed, the result would be wrong. It would then only increment variable j when the value of *char* is "a". That would be wrong because we need j to indicate the position in the current line that is being examined. You should modify the program, making this indentation error, to see exactly what goes wrong.

Similarly, the last statement in the program:

```
print "Character 'a' appears in the input file %d times" % counta
```

is not indented at all. It is aligned with the first "for" in the program. That means that this print statement is *not* in any of the three blocks. And that is just what we want, because we only want that print statement to execute *once*, after all the characters in all the lines have been examined.

Critical Exercise Learn by doing. Experiment with indenting the print statement (leaving it as the last statement in the program) to see what happens in each case. There are five indentations you should try: indent so that the start of the print statement aligns with the start of the first "for"; so that it aligns with the second "for"; so that it aligns with the "if"; so that it aligns with the first "j"; so that it aligns with none of those positions. Some of those indentations will lead to syntactic error, where Python will yell at you; but some will lead to logical errors – Python will be happy, but you will not be. Logical errors are the hardest to figure out and fix. And sometimes, you don't even realize that the program is getting the wrong answer.[14]

Central Message In Python, indentation and block structure are absolutely *crucial* in conveying your logic, i.e., your solution method, to the computer. In fact, block structure is similarly crucial in almost all modern programming languages. And, blocks can nest and interact in very complex ways: you can have a series of many successive blocks inside blocks,

[14]Hopefully it doesn't take a plane crash, stolen election, or unintended nuclear missile launch to let you know that something is amiss.

more than just three; or several independent blocks inside other blocks, etc. Most of your programming energy will go into getting your blocks correct.

However, Python differs from most other programming languages in that blocks are specified by indentation structure – most other languages use *explicit* matching symbols to mark the start and end of a block – symbols such as "{" and "}". Indentation of blocks in those languages is not syntactically required, although consistent indentation really helps you (and others) see the logical structure of the blocks.

The syntax of blocks differs in different languages, but their logical meaning and import is the same. So, what you have learned here about the logic of blocks is good preparation for learning many other programming languages in addition to Python.

5 Back to CLgraphfile

Having introduced variables, Python magic, assignment, iteration, blocks and conditionals, lets look again at program "CLgraphfile.py". The heart of that program contains the following statements:

```
constraints = "such that \n\n"
listC = ""
binaries = "binary \n"
index = 0

for line in INFILE:
    index = index + 1
    listC = listC + "+ C(%d)"  % index
    binaries = binaries + "C(%d)\n" % index

    j = 0
    for char in line:
       j = j + 1
       if char == "0" and index < j:
          string = "C(%d) + C(%d) <= 1\n" % (index,j)
          constraints = constraints + string

INFILE.close()
OUTFILE.write("Maximize \n")
```

```
OUTFILE.write(listC + "\n")
OUTFILE.write (constraints)
OUTFILE.write (binaries)
OUTFILE.write ("end")
OUTFILE.close()
```

The block structure of this code fragment is the same as in *testpy4*. It has three nested blocks, the first two are attached to "for" statements, and the third is attached to the "if" statement. The first block reads one line at a time from the file name held in variable *INFILE*, assigning it to the variable *line*, and incrementing "index" each time. The next statement assigns something to the variable "listC", but what? There are some unfamiliar elements here. Before the start of the first block, we see that "listC" is assigned the empty string, so *listC* is a variable intended to hold a string.

We can then guess that:

```
listC = listC + "+ C(%d)"  % index
```

starts with the current value in *listC*, does something with or to it, and then assigns the result back to *listC*. Also, the statement has symbols "%d", and symbol "%" followed by a variable name, as we saw earlier in a print statement. If these elements work the same in this assignment statement as they do in a print statement, then the effect is to replace "%d" with the value of the variable *index*. In fact, that is what happens. The final unfamiliar element of this statement is the plus sign, "+", that follows "listC" on the right side of the statement. What is being added?

The answer is that two *strings* are added together – but not numerically, since that would make no sense. Rather, the two strings are *concatenated*. That is, the second string is added to the end of the first string. The symbol "+" is used both to numerically add two numbers, and is also used to concatenate two strings. Context tells Python which use is intended: when the two variables to the right and left of the "+" symbol each hold a number, then "+" means *arithmetic add*; but when the two variables each hold a string, then "+" means *concatenation*.[15]

For an example, lets suppose that variable *index* has the value 4. Then, when the above statement is executed by Python, the number 4 will replace

[15]Just to confuse matters, the plus symbol inside the quotes does not specify an addition or a concatenation. It is a part of the string that is being built up. That string will be part of the concrete ILP formulation that is output, as we will see.

the symbols "%d" to create the string "C(4)", and the current string held in variable *listC* will be concatenated together with the string "+ C(4)" (without the quotes).

Now, the statement:

```
listC = listC + "+ C(%d)"  % index
```

is in the first block of the program, so it will be executed once for each line in the input file, and the value held in *index* will be one larger in each iteration. For example, if there are five lines in the input file, the final string held in *listC* will be:

```
C(1) + C(2) + C(3) + C(4) + C(5)
```

And this is what will be printed when the Python program is run and string *listC* is printed. We will see later that this string will be the main part of the objective function for the concrete ILP formulation.

The next line The next statement in the program is:

```
binaries = binaries + "C(%d)\n" % index
```

which is similar to the statement just before it. The variable "binaries" was initially set to the string

```
binary \n
```

Each time the statement above is executed, the value of *binaries* is concatenated with another string, "$C(i)$", where "i" the current value of variable *index*. Those characters are then followed by "`\n`", which will cause Python to move to a new line each time it prints the value of a "$C(i)$". So, the final value of variable *binaries* will be the string:

```
binary \nC(1)\n C(2)\n C(3)\n C(4)\n C(5)\n
```

Later, near the end of program *CLgraphfile*, when the value of variable *binaries* is printed, or written to a file, that string will be output as:

```
binary
C(1)
C(2)
C(3)
C(4)
C(5)
```

5.1 Continuing our examination of CLgraphfile

Now we come to the most critical part of program *CLgraphfile*. Recall that the logic behind the abstract ILP formulation for the maximum clique problem (discussed in Section 2.2 of the book) is that the concrete ILP formulation must contain the inequality:

```
C(i) + C(j) <= 1
```

for *every* pair of nodes (i, j) which are not connected by an edge. So the Python program *CLgraphfile* must create such an inequality for each pair of nodes (i, j) that are *not* neighbors in the input graph. The program does this in the following code fragment:

```
j = 0
for char in line:
   j = j + 1
   if char == "0" and index < j:
      string = "C(%d) + C(%d) <= 1\n" % (index,j)
      constraints = constraints + string
```

This code fragment is inside block 1. Hence this entire code fragment will be executed once after each line is read in from the input file. But note that the code fragment contains two blocks of its own, which we next examine.

The second block of the program (which is inside the first block) examines each character in the string held in variable *line*. The first statement in the block is executed after each new character is extracted from the string. That statement increments variable *j*, which was set to 0 before the start of the second block; hence the value of *j* specifies the position in the string of the current character being examined.

Next, the *if* statement checks whether the character held in variable *char* is "0" *and* whether the value in variable *index* is less than the value in *j*. The word "and" in the "if" statement means that the whole "if" statement will be evaluated to "true" *if and only if* both conditions (char == "0", and index < j) evaluate to true. The word "and" is a *logical operator* – its meaning is built into Python.[16] Python has several built-in

[16]The word "or" is also a logical operator built into Python. Its meaning is that the whole "if" statement would be evaluated to "true" if *one or both* of the two conditions are evaluated to "true". If, by mistake, we had used "or" instead of "and" in this "if" statement, Python would not object, since the statement would still be grammatically permitted, but the program would not behave the way we want it to. Try it out, and see for yourself.

logical operators, which you can learn about in a general text on Python. Here, we only use the "and" operator.[17]

That attached block When the "if" statement evaluates to *true*, the two statements in the attached block are executed. The first one creates the string

```
C(vin) + C(vj) <= 1\n
```

where *vin* denotes here the current value held in variable *index*, and *vj* denotes the current value held in *j*. For example, if the value in *index* is 4 and the value in *j* is 13, then the string created would be:

```
C(4) + C(13) <= 1\n
```

That created string is then assigned to the variable "string"[18] The second statement in the block concatenates the string in variable *string* to the string held in variable *constraints*. So, *constraints* accumulates, as one long string, all of the inequalities that are created in a concrete ILP formulation. Later, that string will be printed in its correct location in the concrete ILP formulation.

You might ask why I have used the variable *string*. Wouldn't it be more direct and compact to replace the two statements in the block with the single statement:

```
constraints = constraints + "C(%d) + C(%d) <= 1/\n" % (index,j)
```

Yes, I could do that, but I prefer to break up long statements so I can see better what is going on in the program. Also, having intermediate steps and additional variables helps with the critical task (in most programs) of debugging.

[17]Sometimes logical operators are written in all capital letters to make it easier (for humans) to see the logic of the program.

[18]Again, I use the word "string" because it reminds me of its purpose in the program. It has no intrinsic meaning in Python. I could have called it "waterfall", but that would have been silly.

More explanation of the "if" The purpose of checking whether the character is "0" in the "if" statement should be clear from the logic of abstract ILP formulation for the maximum clique problem, since a "0" in the input matrix (which encodes the edges that are in the input graph) means that there is *no* edge between the node specified by the value of variable *index*, and the node specified by the value of variable *j*. The purpose of checking whether the value in *index* is less than the value in *j* is to make sure that the next statement is only executed once for any pair of positions in the string. Otherwise, the concrete ILP produced by *CLgraphfile* would contain the inequality:

```
C(i) + C(j) <= 1
```

and also the inequality:

```
C(j) + C(i) <= 1
```

for each pair of nodes i, j that are not adjacent in the input graph.[19] You might modify the "if" statement by removing that second check, and see for yourself how the concrete ILP formulation changes.

5.2 Putting it together

OK, so now we have explained how *CLgraphfile* accumulates a string, held in the variable *listC*, containing all of the C variables, with the plus sign "+" between each successive pair; how it accumulates a string, held in the variable *binaries*, containing all of the C variable, with `"\n"` between each successive pair; and how it accumulates a string, held in variable *constraints*, which starts with "such that `\n`" and follows with all the inequalities (constraints) needed for a concrete ILP formulation of the maximum clique problem.

The final outputs Finally, the program writes out the entire concrete ILP formulation to the file whose name is held in variable *OUTFILE*. The Python syntax to write a string to a file is:

```
FILE-VARIABLE.write(string or statement evaluating to a string)
```

[19]Such redundancy is actually fine. Gurobi (and other ILP solvers) will identify the redundancy and remove one of the inequalities in the *preprocessing* phase, but since the concrete ILP formulations tend to be large, making it harder to read it, my choice is to avoid the redundancy in the first place.

The specific statements in *CLgraphfile* are:

```
OUTFILE.write("Maximize \n")
OUTFILE.write(listC + "\n")
OUTFILE.write (constraints)
OUTFILE.write (binaries)
OUTFILE.write ("end")
```

In these "write" statements we see examples of different ways that the "string or statement evaluating to a string" requirement is met.

Finally, the statement:

```
print ("The ILP file is: %s \n" % sys.argv[2])
```

prints the name of the ILP file that is created by this Python program, and the statements

```
INFILE.close()
OUTFILE.close()
```

close the files whose names are held in variables *INFILE* and *OUTFILE*.

5.3 An alternative approach

In program *CLgraphfile*, all of the constraints are concatenated into a single string, called "constraints". That string has a new-line symbol between each successive constraint, so that when the string is printed, each constraint appears on its own line. Similarly, the set of ILP variables that will appear in the objective function is accumulated in a single string, "listC", as are the ILP variables that are listed below the word "binary" in the ILP formulation.

I like this style of programming to construct concrete ILP formulations, but it may seem more natural to use Python "lists" instead of strings, and discussing this alternative approach is a way to introduce a bit more of Python.

Python lists A *list* in Python is just what it sounds like, an ordered collection of elements (things) – numbers, strings, or values specified by Python variables – for now. The elements in a list are enclosed by left and right *brackets*, and are separated by commas. For example, the statement:

```
binaries = ["C(1)", "C(2)", "C(3)"]
```

assigns a list of three strings to the variable *binaries*. Note that each string is enclosed in quotes.

When the value of variable *binaries* is printed, the result shows explicitly that the value is a list, and that contains strings. Adding the statement:

```
print binaries
```

to the Python assignment statement above, yields:

```
['C(1)', 'C(2)', 'C(3)']
```

A list can have a mixture of strings, numbers and values held in a variable. For example, the output of the following code fragment:

```
height = 10
alist = ["tom", 53, height]
print alist
```

is:

```
['tom', 53, 10]
```

If we want to individually access or print *all* of the values in a list, we can use the "for, in" statement:

```
for element in alist:
    print element
```

where "element" is a variable. The effect of this "for, in" statement is similar to the "for, in" statements we saw earlier, although now the successive values are the elements in a list, rather than lines from a file, or characters in a string. The above code fragment produces:

```
tom
53
10
```

Notice that each element in the list is printed on its own line, even without a `"\n"` in the print statement. Also, note that the string "tom" is printed without quotes.

Exercise Change the "for" statement above to:

```
for element in [alist]:
```

to see what is printed. Then try to explain the result.

If we want to access an individual element in a list, we use the name of the list, followed by square brackets that enclose the position of the element we want; or a variable whose value is the position. For example,

```
alist[2]
```

specifies a single element in the list *alist*. But which one exactly? It is the *third* element, the number 10. Why the third and not the second? Because Python, as many other programming languages do, begins indexing with zero (Yuck!). So *alist[0]* is the first element in *alist*, with value "tom", and *alist[1]* is the second element in *alist*, with value 53.

Illustrating how to use indexing to access and print elements in *alist*, the following code fragment accomplishes the same result as the "for, in" statement discussed above.

```
for i in range(0, len(alist)):
    print alist[i]
```

In addition to the index of a list, there are two new syntactic features of Python that are used in this code fragment. First, *len* is a built-in Python *function* which, applied to *alist*, finds the length of the list *alist*, i.e., number of elements in *alist*. In this example, the length is 3. Second, the built-in Python *range* function in:

```
range(0, len(alist))
```

generates a list of integers starting with 0 and ending with *len(alist) minus 1*. So, for our example,

```
for i in range(0, len(alist)):
    print alist[i]
```

is equivalent to:

```
for i in [0,1,2]:
    print alist[i]
```

but the use of *len(alist)* is more general since it can accommodate any length that list *alist* might be. That length might not even be known at the time that the program is written. Of course, the original syntactic approach, i.e.,

```
for element in alist:
    print element
```

is similarly general and easier to write. Still, these examples illustrate the use of list indexing, which is often the Python syntax we need. You should try out all of the variations we have described here to see exactly how they work, and see for yourself what limitations each approach has.

5.3.1 Adding an element to an existing list

Each of the lists illustrated above were created with a single assignment statement. But in most applications, lists are built up over time, during the execution of the program. Typically, one wants each new element to be added to the end of the growing list. This is achieved through the use of what is called a "string method" in Python, and in particular, the method called "append". The syntax for using a method is to write the name of the variable that holds a string, followed by a *dot*, followed by "append", followed, in parenthesis, by the item that is to be added to the list.[20] For example, we can add an element to *alist* as follows:

```
alist.append(48)
```

Then, the code fragment:

```
for element in alist:
    print element
```

will print:

```
tom
53
10
48
```

[20]Notice that our use of "write", above, is of this form, and "write", in fact, is also a Python *method*.

Before appending any new items to a list, the variable holding the list must already hold a list, as shown above, or have been explicitly assigned the empty list. For example, in the following code fragment, the first statement assigns the empty list to variable "blist", and then the string "susan", and the number 15, are appended to the list. The end result is that *blist* holds a list consisting of two elements.

```
blist = []
blist.append("susan")
blist.append(15)
```

5.4 Using lists in *CLgraphfile*

OK, so now that we have seen a bit about Python lists and how to access individual elements in a list, we explain how lists can be used in a modification of the program *CLgraphfile*. In the following code fragment, the three variables *listC, binaries* and *constraints* which each had previously accumulated and held a single (long) string, now each accumulate and hold a list of strings.

```
index = 0
listC = []
constraints = ["\n such that \n"]
binaries = ["binary \n"]

for line in INFILE:
    index = index + 1
    string = " + C(%d)"  % index
    listC.append(string)
    string = "C(%d)\n" % index
    binaries.append(string)

    j = 0
    for char in line:
       j = j + 1
       if char == "0" and index < j:
          string = "C(%d) + C(%d) <= 1\n" % (index,j)
          constraints.append(string)
```

Notice that we reset the variable "string" many times – it is really just a shuttle from the statement where a string is constructed, to the next

statement where the string is used. After that, its value can be changed without harm.

Near the end of the program, instead of writing out three long strings, as done in the original *CLgraphfile*, we write out three lists of strings, as follows:

```
OUTFILE.write("Maximize \n")
for string in listC:
     OUTFILE.write(string)
for string in constraints:
     OUTFILE.write(string)
for string in binaries:
     OUTFILE.write (string)
OUTFILE.write ("end")
```

The use of lists instead of strings is a more standard way to write in Python, but the main reason for converting program *CLgraphfile* in this way is just to introduce strings and discuss their use in Python. We will see a more involved use of lists when we discuss the program *first-RNA* in the next section.

5.5 Summary

Perhaps surprisingly, this little program, *CLgraphfile* contains most of the critical constructs used in Python, and many other programming languages. It contains variables, assignments, iteration, blocks, conditionals, formatted printing, tuples, lists, functions, and methods. Of course, Python contains more *syntactic* choices than what we have introduced here. For example, there are several ways to specify iteration – more than the way we did it here, with the use of a "while" statement. We will see some more Python syntax in our discussion of program *first-RNA*. And there are a few central programming language *concepts*, for example *multi-dimensional lists*, *dictionaries* and *procedures*, that we will not discuss.[21] But again, it is really interesting that such a simple program as *CLgraph-file.py* contains most of the critical constructs in Python and for procedural programming in general.

[21] We have not touched object-oriented programming.

6 RNA folding: More complex programming and more Python syntax

In this section we discuss the Python program for simple RNA folding, (*first-RNA*), discussed in Chapter 6 of the book. We try to make the Python programming as simple as possible. As a consequence, this program generates some unneeded ILP variables, and some unneeded inequalities, but these do no harm. They will be cleaned up by the Gurobi preprocessor.

Remember that the problem is to find a legal pairing of characters in an RNA string that obeys three critical constraints. First, each pair of characters in a pairing must be *complementary*: A and U can pair; C and G can pair. Second, each character, at a given position in the RNA string, can be in at most one pair in a pairing. Third, the set of pairs in a pairing must be *non-crossing*: There cannot be four positions in the RNA string $h < i < j < k$, where the character at position h is paired with the character at position j, and the character at position i is paired with the character at position k. Then, the objective is to find as many pairs as possible, obeying the above three constraints. Recall that the ILP variable $P(i, j)$ will be set to 1 in the optimal ILP solution, if the character in position i pairs with the character in position j of the RNA string. Otherwise, it is set to 0.

Program *first-RNA* is shown below.

```
# first-RNA.py  # for python 2.7
#
# Python program to generate the inequalities for the ILP
# formulation for the simple RNA folding problem,
# discussed in Chapter 6, Section 6.1.1.
#

#
OUT = open ("RNA1.lp", 'w')

# Collect information from the user
print ("Will you type in the RNA sequence, or read it from file ‘randomstring’?" )

inchoice = raw_input("Type 't' for typed input, 'f' for file. ")
if (inchoice == 't'):
   RNA = raw_input("Write an RNA sequence (using A,U,C,G, and hit return \n")
else:
   IN = open("randomstring", 'r')
   RNA = IN.readline().strip()

print "You input the sequence: %s" % RNA

length = len(RNA)

# Generate the objective function

OUT.write ("Maximize \n")
for i in range(1, length):
  for j in range(i+1, length+1):
         OUT.write ("+ P(%d,%d) \n" % (i,j))
OUT.write ("such that \n")

# The following segment generates the ILP inequalities to ensure
# that only complementary nucleotides pair.

for i in range(1, length):
  for j in range(i+1, length+1):

           if (RNA[i-1] == 'A') and (RNA[j-1] != 'U'):
                   OUT.write ("P(%d,%d) = 0 \n" % (i,j))

           if (RNA[i-1] == 'U') and (RNA[j-1] != 'A'):
                   OUT.write ("P(%d,%d) = 0 \n" % (i,j))

           if (RNA[i-1] == 'C') and (RNA[j-1] != 'G'):
                   OUT.write ("P(%d,%d) = 0 \n" % (i,j))

           if (RNA[i-1] == 'G') and (RNA[j-1] != 'C'):
```

```
                OUT.write ("P(%d,%d) =  0 \n" % (i,j))

# The following segment generates the ILP inequalities to ensure
# that each position is paired to at most one other position

for i in range(1,length):
   inequality = ""

   for j in range(1, i):
            inequality = inequality + " + P(%d,%d)" % (j,i)

   for j in range(i+1, length+1):
            inequality = inequality + " + P(%d,%d)" % (i,j)

   inequality = inequality + ' <= 1'
   OUT.write (inequality)
   OUT.write ("\n")

# The following segment generates the ILP inequalities to ensure
# that no pairs cross.

for h in range(1, length - 2):
    for i in range(h+1, length - 1):
        for j in range(i+1, length):
            for k in range(j+1, length+1):
                OUT.write ("P(%d,%d) + P(%d,%d) <= 1 \n" % (h,j,i,k))

# Write out the list of binary variables
OUT.write ("binary \n")
for i in range(1, length):
   for j in range(i+1, length+1):
        OUT.write ("P(%d,%d) \n" % (i,j))

OUT.write  ("end \n")
print "The ILP file is RNA1.lp"
```

We will not explain each statement in program *first-RNA*, but only those statements containing Python elements that were not in program *CLgraphfile*. Any statement not explained should be understandable after understanding the discussion of *CLgraphfile*.

Let's look first at the following fragment of program *first-RNA*:

```
inchoice = raw_input(" 't' for typed input, 'f' for file 'randomstring'.")
```

```
if (inchoice == 't'):
   RNA = raw_input("Please type an RNA sequence and hit return \n")
else:
   IN = open("randomstring", 'r')
   RNA = IN.readline().strip()
```

The purpose of this code fragment is to ask the user whether they will input an RNA string by typing one out at the keyboard, or if they will input it from a file called "randomstring". The second line in this fragment has a Python construct that we have not encountered before. That line has the function `raw_input` followed by a string encolsed in parenthesis. To the left of `‘‘raw_input"` is an equality sign and the variable "inchoice". Function `raw_input` is built into Python 2.7 (it is not in Python 3, and this is one of few places where the difference in Python versions will concern us). When function `raw_input` is executed, the string enclosed in parentheses is printed to the monitor (screen), and the program waits for input typed on the keyboard. That input is interpreted as a string and assigned to the variable on the left of the equal sign. So, in this specific example, the following will be printed to the monitor:

```
Will you type in the RNA sequence, or read it from file ‘randomstring’?
‘t’ for typed input, ‘f’ for file ‘randomstring’.
```

Now the program pauses and waits for input (ended with a return key) from the keyboard. If the user is interacting as expected, they will either type ‘t’ or ‘f’, followed by the return key. We will see soon what happens if they input something different.

The next two statements in this fragment use Python syntax that has been explained already. But the line after those is:

```
else:
```

which is aligned with the "if" two lines before it. It is part of a Python "if, else" construct. The meaning of this construct is that if the condition in the "if" statement is *true*, then the block attached to that "if" statement will be executed, and the block attached to the "else" statement will not be executed. But if the condition in the "if" statement is false, then the block attached to it will not be executed, and the block attached to the "else" statement will be executed. Such "if, else" constructs are very useful and common in computer programs.

So, let's look at what is in the block attached to the "else". The first statement is an "open" statement of the kind we have discussed already. The next statement looks like it might be of the form we discussed earlier when discussing Python *methods*, but instead of having a single dot, it has two dots, and there is nothing inside the two parentheses. So, what does this statement do? We will first consider the shorter statement:

```
RNA = IN.readline()
```

This is an application of the Python method "readline" to the file whose name is held in variable *IN*. The effect is to read the next *unread* line in that file (which is "randomstring"), and assign that line (as a string) to the variable called "RNA". The line starts at the start of the file, if there have been no previous reads, or one character after the point where the file was last read. That line ends where a newline character (`"\n"`) is first encountered, or at the end of the file.

Now, there are ways to modify the behavior of "readline", and if any are desired, they would be specified as parameters inside the parentheses. But in "first-RNA", no modifications are desired, so the parentheses have nothing in them. Even with nothing inside them, Python requires them.

The full statement Now, we return to the full statement:

```
RNA = IN.readline().strip()
```

which contains two Python methods: *strip*, in addition to *readline*. Based on what we have seen in the prior uses of Python methods, we may infer that *strip* is applied to the result of whatever comes before the dot in *.strip()*. What comes before is a line that is read in from the file whose name is held in the variable *IN*. So *strip* applies to that string.

But what exactly does *strip* do? It removes any "whitespaces" at the beginning and at the end of the string. A *whitespace* in Python is either a space or a tab symbol. So, the string that is assigned to variable RNA will have no spaces or tabs at its front or at its end. And, this is very desirable since we only want the RNA folding program to work on the real RNA string and not any characters around it.

len The next new element of Python in *first-RNA* is the function "len", which appears in the statement:

```
length = len(RNA)
```

Function "len" is a built-in Python function which is applied to a string, and returns the *length* (number of characters) in the string. So, the variable *length* is assigned the number of characters in string held in *RNA*, which is the string just read in from the file "randomstrings".

range The next code fragment generates the objective function in the ILP formulation. For that, the program needs to generate every $P(i, j)$ variable, for $i < j \leq length$, and create a string that sums those variables together. Note that the value of i should be at most $length + 1$, since j must be larger than i. The code fragment is:

```
OUT.write ("Maximize \n")
for i in range(1, length):
  for j in range(i+1, length+1):
         OUT.write ("+ P(%d,%d) \n" % (i,j))
```

The new Python in these statements is the built-in function *range*, which produces a list of all the integers from a beginning integer to an ending integer. Consider:

```
range(1, length)
```

You might guess that the starting integer is 1, and you would be right. You might also guess that the ending integer is the value in variable *length*. That would be sensible, but wrong. Instead, the ending integer is *one less* than the value in *length*. For example, if the value in *length* is 5, then *range(1, length)* generates the list $(1, 2, 3, 4)$. So if you want to generate a list that contains all the integers from 1 to one less than length of the string in *RNA*, you call function *range* with the parameters $(1, length)$. And if you want to generate a list that contains all integers from the value of $i + 1$ to the value in *length*, you call function *range* with the parameters $(i + 1, length + 1)$.

Now, you may ask why the *range* function in Python only generates integers to one below the value of its second parameter. Why doesn't it do the more sensible thing of generating integers out to the actual value of its second parameter. Well, you got me - I don't know! I would not have designed Python with that "feature", but I didn't design Python.

Only complementary nucleotides can pair The next code fragment we consider generate the inequalities that ensure that only complementary nucleotides are allowed to pair.

```
for i in range(1, length):
  for j in range(i+1, length+1):

          if (RNA[i-1] == 'A') and (RNA[j-1] != 'U'):
                  OUT.write ("P(%d,%d) = 0 \n" % (i,j))

          if (RNA[i-1] == 'U') and (RNA[j-1] != 'A'):
                  OUT.write ("P(%d,%d) = 0 \n" % (i,j))

          if (RNA[i-1] == 'C') and (RNA[j-1] != 'G'):
                  OUT.write ("P(%d,%d) = 0 \n" % (i,j))

          if (RNA[i-1] == 'G') and (RNA[j-1] != 'C'):
                  OUT.write ("P(%d,%d) =  0 \n" % (i,j))
```

The only new Python syntax in this code fragment is the symbol "!=". That symbol means "not equal". So, for example, if the character in position $i-1$ in the string held in variable *RNA* is "A", and the character in position $j-1$ is *not* "U", then those two positions are not allowed to form a pair. So we want the ILP formulation to contain the equality

$$P(i,j) = 0$$

.

You may ask why these statements have indices $i-1$ and $j-1$ instead of i and j. It is because I use 1 as the first index of the RNA string. That is the sane and sensible thing to do – the first element of anything is element number 1, isn't it? But, at some point in the history of computer programming, someone decided that the index should be 0, and most subsequent languages (and Python in particular) followed this awful convention. So, the first element in a Python list is element 0. Hence, to access my character i of the RNA string, you have to access character $i-1$ of the Python string.[22]

[22]Wouldn't life be easier if I just gave in and joined the zero-first convention? No! Never!

Onward The next segment of program *first-RNA* creates the ILP inequalities to ensure that no position in the RNA string is paired with more than one other position. This code fragment has no new Python elements, so you should be able to figure out how it works without my help.

The next segment creates the ILP inequalities to ensure that the pairing specified by the $P(i, j)$ variables with values 1, is a *non-crossing* pairing. Again, it has no new Python syntax, and you should be able to figure out how it works. I will only point out that it has *four* nested blocks (each preceded by a "for" statement), and the effect is that these blocks generate all possible combinations of values for variables h, i, j, k such that $1 \leq h < i < j < k \leq length$.

Finally, the program ends by writing out, to the file "RNA1.lp", all the remaining ILP inequalities, and writing to the monitor to inform the user that the concrete ILP formulation is there.

7 What did we leave out about Python?

Lots! We left out lots about Python syntax. That is partly why there are thousand page books of Python. And we left out some important elements of procedural programming in general. But, as stated earlier, it is very surprising (and pleasing) how much of Python, and of programming, is illustrated in just these two simple programs. Master these programs and you are well on your way to mastering procedural Python programming.[23]

I will end by mentioning one additional important element of programming in Python (and other languages), that of *multi-dimensional* lists. The simplest is a two-dimensional list, where the elements of the list are not numbers, or strings, or variables (as discussed here), but are *lists*. So a two-dimensional list is a *list of lists*. This is illustrated, and explained in the comments, in the RNA-folding program *fourth-RNAf.py*.

[23] Again, we are completely ignoring *object-oriented programming*.