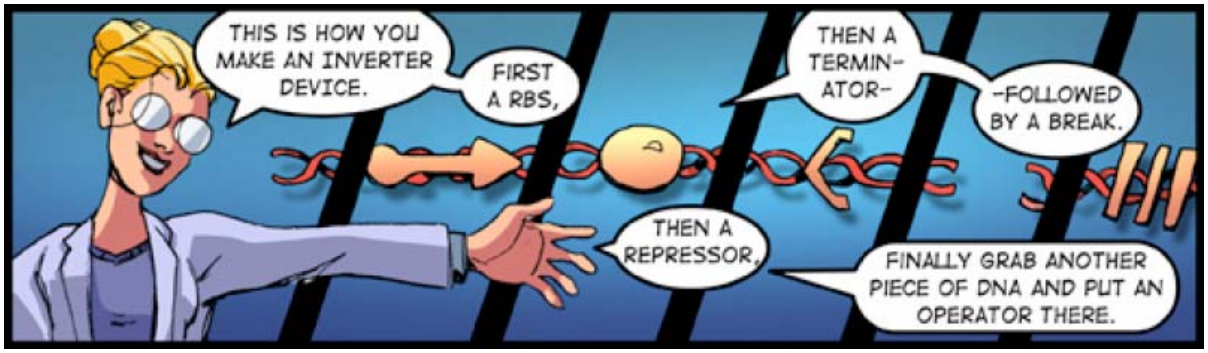
# BE.180 – Homework Assignment #2

*Released on March 2; Due at 5pm on Thursday, March 9*




We have discussed the following levels of abstraction in class, listed here with an example of each in parentheses:

Systems (ring oscillator)
↕
Devices (inverter)
↕
Parts (ORF)
↕
DNA (ATGC…)

In the first homework assignment, you manipulated Parts and their DNA sequences and became comfortable working at this abstraction barrier. Now we will advance to the next level of the abstraction hierarchy to construct Devices. You will practice making inverters (a type of Device) from various Parts (composed of DNA). **For this assignment, write code to construct a library of all possible inverters from a dictionary of Parts, and then search through all of the inverters to find the inverter with the lowest GC-content.** Before reading on, think about how you might do this.

By the end of this assignment, you will have practiced:

1. manipulating complex data structures (a dictionary of dictionaries)
2. writing functions
3. creating classes and using objects (taking advantage of the fact that Python is an object-oriented programming language)

*Note: Some parts of this assignment contain rhetorical questions that are for your own benefit, and the answers do not need to be submitted.*

Part I. Manipulating a dictionary of dictionaries (30 points)

A. You will be provided with a dictionary of Parts called `parts.dict`. Open the file with a text editor such as Pico and note the structure of the dictionary. As you will see, this dictionary contains the keys: `RBSDict`, `ORFDict`, `terminatorDict`, and `promoterDict`. For each of these keys, the value is a nested dictionary of Part names as keys and corresponding DNA sequences as values. This dictionary was constructed from the Registry of Standard Biological Parts (http://parts.mit.edu). Go to this webpage to learn about the Parts in the dictionary. You may also refer to Chapter 2 of the comic strip to read about inverters.

Create a new Python script called `yourAthenaName_2.py`. Read in the `parts.dict` file using `eval(open("DictionaryFileName").read())` and store the contents to a dictionary called `myDict`.

B. Recall that you can access all the keys of a dictionary using `dictionaryName.keys()`, and the value corresponding to a specific key by `dictionaryName["key"]`. How many key-value pairs are there in each of the four subdictonaries (in other words, the number of Parts of each type)? What is the DNA sequence of the ORF corresponding to Part C0051?

C. Now you will construct a library of all valid inverters. Each inverter will contain the DNA sequences of four Parts, in this order: one RBS, one ORF, one terminator, and one promoter. Refer to lectures notes from February 14th and 16th and the comic strip to see why you might want them in this particular order. A valid inverter must have a biologically compatible ORF and promoter (see lecture notes). `map.dict` is a dictionary of compatible ORFs and promoters constructed from http://parts.mit.edu. In this dictionary, the key is the ORF Part name, and the value contains the compatible promoter Part name.

Write a loop that will create this library of valid inverters. Read in `map.dict` and store it as a dictionary called `map`. Use the mapping in this dictionary to ensure that you only create valid inverters. For each valid inverter, store the name of the inverter as the key and the concatenated DNA sequence as the value in a dictionary called `invertersDict`. The name of each valid inverter should be constructed as follows: `RBSPartName.ORFPartName.terminatorPartName.promoterPartName` (e.g. `B0034.C0051.B0015.R0051`). How many total inverters (valid and invalid) would you expect to find, based

on the number Parts there are of each type? Did you end up with the correct number of *valid* inverters, when you take into consideration the compatibilities provided in `map.dict`?

Part II. Writing functions (10 points)

*Introduction to functions:*

Functions allow the programmer to store valuable procedures and use them frequently. They provide a form of abstraction, since you do not need to know the inner workings of a function in order to use it. All you need to know are the type and number of arguments the function accepts, and what it returns. Functions can accept multiple arguments, but the order in which you list them in the function call and in the function definition must be the same. Note that the names of the arguments in the function call do not need to match the names of the arguments in the function definition.

Functions are defined at the top of your Python script and are called in the main program, which is below all functions and classes. In Python, a function block is denoted with the command `def`. Any output of the function that you will want to access in the main program needs to be returned within the function. Here is an example of a simple function called `helloFunction` and its output:

```
>>> def helloFunction(argument):
...     if argument=="Hi":
...         return argument + " there"
...     else:
...         return "Why don't you say Hi?"
...
>>> output1=helloFunction("Hi")
>>> output1
'Hi there'
>>> output2=helloFunction("Drew Endy")
>>> output2
"Why don't you say Hi?"
```

A. You will now write a function that calculates the GC-content of each inverter in the dictionary of valid inverters that you constructed in Part I. GC-content is the proportion of bases in a DNA sequence that are cytosines or guanines. Knowing the GC-content of a particular DNA sequence may be useful since sequences with a high GC-content are more resistant to denaturation by high temperatures.

Write a function called `getGCcontent1`, which takes in `invertersDict` as its only argument and returns a list containing the GC-content of each valid inverter. Call the function from the main program and store the output as `GClist1`.

Part III. Creating classes and using objects (60 points)

*Introduction to classes and objects (from java.sun.com/docs/books/tutorial/java/):*

In the real world, you often have many objects of the same kind. For example, your bicycle is just one of many bicycles in the world. Using object-oriented terminology, we say that your bicycle object is an instance of the class of objects known as bicycles. Bicycles have some state (current gear, current cadence, two wheels) and behavior (change gears, brake) in common. However, each bicycle's state is independent of and can be different from that of other bicycles. When building them, manufacturers take advantage of the fact that bicycles share characteristics, building many bicycles from the same blueprint. It would be very inefficient to produce a new blueprint for every bicycle manufactured.

In object-oriented software, it is also possible to have many objects of the same kind that share characteristics. Like bicycle manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects. A software blueprint for objects is called a class.

The class for our bicycle example would declare the instance variables necessary to contain the current gear, the current cadence, and so on for each bicycle object. The class would also declare and provide implementations for the instance methods that allow the rider to change gears, brake, and change the pedaling cadence. After you've created the bicycle class, you can create any number of bicycle objects from that class. When you create an instance of a class, the system allocates enough memory for the object and all its instance variables. Each instance gets its own copy of all the instance variables defined in the class.

Objects provide the benefit of modularity and information-hiding. Classes provide the benefit of reusability. Bicycle manufacturers use the same blueprint over and over again to build lots of bicycles. Software programmers use the same class, and thus the same code, over and over again to create many objects.

A. In this section, you will use some of your code from Part I to create a class and construct inverter objects. To create the class Inverter, insert this code at the top of your Python script:

```
class Inverter:
   def __init__(self, thisRBS, thisORF, thisTerminator, thisPromoter):
      self.RBS=thisRBS
      self.ORF=thisORF
      self.terminator=thisTerminator
      self.promoter=thisPromoter

   def getORFLength(self):
      return len(self.ORF[1])
```

The `__init__` function is the constructor function for the class Inverter, and is called whenever a new object is created. To create each new Inverter object, you will need to specify the appropriate arguments, which will be fed into the constructor. Note that the constructor that we defined above takes in five arguments. The first argument, `self`, automatically references the object you are currently working with; therefore, you only need to pass in the other four arguments when creating a new object. `self` refers to a specific object, so `self.foo` accesses foo, which can be a variable or function associated with that object. For example, the function `getORFLength` in class Inverter returns the length of the DNA sequence of the ORF that is associated with a specific Inverter object (`self.ORF[1]` is used to access the second item in the two-item `self.ORF` list, defined below).

Now, as you did in Part I.C, you should write a loop to construct all possible valid inverters using `myDict`. Here, instead of saving each one to a dictionary, create a new Inverter object for each new valid inverter as follows: `newInverter=Inverter(RBSInfo, ORFInfo, terminatorInfo, promoterInfo).` `partInfo` should be a two-item list, where the first item is the Part's name (e.g. `B0034`), and the second item is the Part's DNA sequence. You should also save all of the objects you create to a list called `inverters`.

B. To be able to access the name and DNA sequence of each Inverter object, you will now write two functions within the Inverter class in order to do this. This should be the structure of your two functions:

```
def getName(self):
   more code can be added here
   return the concatenated name of the inverter (e.g. B0034.C0051.B0015.R0051)

def getSeq(self):
   more code can be added here
   return the concatenated DNA sequence of the inverter
```

In the main program, loop through the list `inverters` and store the name and DNA sequence for each inverter to the lists `nameList` and `seqList`, respectively, using the two functions you just wrote.

C. Use pieces of your code from Part II.A to write a new function in your Inverter class called `getGCcontent2`, which will find the GC-content of each Inverter object by making use of the `getSeq` function. Fill in this function:

```
def getGCcontent2(self):
   more code can be added here
   return the GC-content for each Inverter object
```

In the main program, loop through the list `inverters` and store each inverter's GC-content to the list `GClist2` using the function you just wrote.

D. Finally, you should find the inverter with the lowest GC-content. Create a function outside of the class Inverter (think about why this makes sense) as follows:

```
def getLowestGC(inverters, GClist2):
   more code can be added here
   return the part name corresponding to the inverter with the lowest GC-content
```

Call this function from the main program and store the output to the variable `lowestGC`. End your code with this line: `print "The inverter with the lowest GC content is ", lowestGC.`