Functions

Genome 559: Introduction to Statistical and Computational Genomics

Elhanan Borenstein
A quick review

- **Dictionaries:**
  - key:value pairs
  - a.k.a. hash tables, lookup tables
  - Examples:
    - Word and definition
    - Name and phone number
    - Gene name and score
    - Username and password
  - Dictionaries are useful when you want to look up some data (value) based on a key
  - Each key can appear only once

- **Standard I/O**
Note: dictionary and list access times

- Accessing a list by index is very fast!
- Accessing a dictionary by key is very fast!
- Accessing a list by value (e.g. list.index(myVal) or list.count(myVal)) can be SLOW.

(index points directly to position in memory)
import sys
matrixFile = open(sys.argv[1], "r")
matrix = []  # initialize empty matrix
line = matrixFile.readline().strip()  # read first line stripped
while len(line) > 0:  # until end of file
    fields = line.split("\t")  # split line on tabs, giving a list of strings
    intList = []  # create an int list to fill
    for field in fields:  # for each field in current line
        intList.append(int(field))  # append the int value of field to intList
    matrix.append(intList)  # after intList is filled, append it to matrix
    line = matrixFile.readline().strip()  # read next line and repeat loop
matrixFile.close()

for row in matrix:  # go through the matrix row by row
    for val in row:  # go through each value in the row
        print val,  # print each value without line break
    print ""  # add a line break after each row
In theory, what you know so far allows you to solve any computational task ("universality")

So ... why don’t we stop here?
most real-life tasks will be (very) painful to solve using only what you know so far ...
What are we missing?
What are we missing?

- A way to generalized procedures ...
- A way to store and handle complex data ...
- A way to organize our code ...
- Better design and coding practices ...
Functions
Why functions?

- Reusable piece of code
  - write once, use many times
  - Within your code; across several codes

- Helps simplify and organize your program

- Helps avoid duplication of code
What a function does?

- Takes defined inputs *(arguments)* and may produce a defined output *(return)*

Other than the arguments and the return, everything else inside the function is invisible outside the function *(variables assigned, etc.)*. Black box!

- The function doesn't need to have a return.
- Spoiler: The arguments can be changed and changes are visible outside the function
Jukes-Cantor model

Jukes-Cantor model:

$$D = -\frac{3}{4} \ln \left(1 - \frac{4}{3} D_{\text{raw}} \right)$$

$D_{\text{raw}}$ is the raw distance (what we directly measure)

$D$ is the corrected distance (what we want)
import math

def jc_dist(rawdist):
    if rawdist < 0.75 and rawdist > 0.0:
        newdist = (-3.0/4.0) * math.log(1.0 - (4.0/3.0)*rawdist)
        return newdist
    elif rawdist >= 0.75:
        return 1000.0
    else:
        return 0.0

def <function_name>(<arguments>):
    <function code block>
    <usually return something>
Using (calling) a function

```python
import sys
dist = sys.argv[1]
correctedDist = jc_dist(dist)
```
Once you've written the function, you can forget about it and just use it!
import sys
dist = sys.argv[1]
correctedDist = jc_dist(dist)

AnotherDist = 0.354
AnotherCorrectedDist = jc_dist(AnotherDist)

OneMoreCorrectedDist = jc_dist(0.63)
Using (calling) a function

```python
import sys
dist = sys.argv[1]
correctedDist = jc_dist(dist)

AnotherDist = 0.354
AnotherCorrectedDist = jc_dist(AnotherDist)

OneMoreCorrectedDist = jc_dist(0.63)

# What about ...
jc_dist(0.57)
```
From “In-code” to Function

**Jukes-Cantor distance correction written directly in program:**

```python
import sys
import math

rawdist = float(sys.argv[1])
if rawdist < 0.75 and rawdist > 0.0:
    newdist = (-3.0/4.0) * math.log(1.0 - (4.0/3.0)* rawdist)
    print newdist
elif rawdist >= 0.75:
    print 1000.0
else:
    print 0.0
```

**Jukes-Cantor distance correction written as a function:**

```python
import sys
import math

def jc_dist(rawdist):
    rawdist = float(sys.argv[1])
    if rawdist < 0.75 and rawdist > 0.0:
        newdist = (-3.0/4.0) * math.log(1.0 - (4.0/3.0)* rawdist)
        return newdist
    elif rawdist >= 0.75:
        return 1000.0
    else:
        return 0.0
```

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add a function definition</td>
</tr>
<tr>
<td>Delete - Use function argument instead of argv</td>
</tr>
<tr>
<td>Return value rather than printing it</td>
</tr>
</tbody>
</table>
We've used lots of functions before!

- math.log(value)
- readline(), readlines(), read()
- sort()
- split(), replace(), lower()

- These functions are part of the Python programming environment (in other words they are already written for you).

- Note - some of these are functions attached to objects (and called object "methods") rather than stand-alone functions. We'll cover this later.
Function names, access, and usage

- Giving a function an informative name is very important! Long names are fine if needed:
  ```python
def makeDictFromTwoLists(keyList, valueList):
    def translateDNA(dna_seq):
      def getFastaSequences(fileName):
  ```

- For now, your function will have to be defined within your program and before you use it. Later you'll learn how to save a function in a module so that you can load your module and use the function just the way we do for Python modules.

- Usually, potentially reusable parts of your code should be written as functions.

- Your program (outside of functions) will often be very short - largely reading arguments and making output.
import sys
myFile = open(sys.argv[1], "r")
# make an empty dictionary
scoreDict = {}
for line in myFile:
    fields = line.strip().split("\t")
    # record each value with name as key
    scoreDict[fields[0]] = float(fields[1])
myFile.close()

Use:
scoreDict = makeDict(myFileName)
import sys

def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict

myFileName = sys.argv[1]
scoreDict = makeDict(myFileName)
import sys

def makeDict(fileName):
    myFile = open(fileName, "r")
    myDict = {}
    for line in myFile:
        fields = line.strip().split("\t")
        myDict[fields[0]] = float(fields[1])
    myFile.close()
    return myDict

myFileName = sys.argv[1]
scoreDict = makeDict(myFileName)

Two things to notice here:
- you can use any file name (string) when you call the function
- you can assign any name to the function return

*(in programming jargon, the function lives in its own namespace)*
Sample problem #2

Write a function that mimics the `<file>.readlines()` method. Your function will have a file object as the argument and will return a list of strings (in exactly the format of `readlines()`). Use your new function in a program that reads the contents of a file and prints it to the screen.

You can use other file methods within your function, and specifically, the method `read()` - just don't use the `<file>.readlines()` method directly.

Note: This isn't a useful function, since Python developers already did it for you, but the point is that the functions you write are just like the ones we've already been using. BTW you will learn how to attach functions to objects a bit later (things like the split function of strings, as in `myString.split()`).
import sys

def readlines(file):
    text = file.read()
    tempLines = text.split("\n")
    lines = []
    for tempLine in tempLines:
        lines.append(tempLine + "\n")
    return lines

myFile = open(sys.argv[1], "r")
lines = readlines(myFile)
for line in lines:
    print line.strip()
Challenge problem

Write a program that reads a file containing a tab-delimited matrix of pairwise distances and puts them into a 2-dimensional list of distances (floats). Have the program accept two additional arguments, which are the names of 2 sequences from the matrix, and print their distance.

Here's what the file contents look like:

```
names	seq1	seq2	seq3
seq1	0	0.1	0.2
seq2	0.1	0	0.3
seq3	0.2	0.3	0
```

Be sure it works with ANY matrix file with this format! The file will always be a square matrix of size (N+1) x (N+1). N for each distance and 1 row and column for names.

```
>python dist.py matrixFile seq2 seq3
0.3
```

Hints: use the first line to make a dictionary of names to list indices; your function should return a 2-dimensional list of floats.
I wrote both complex parts as functions; this makes the point that once these are written and debugged, the program is simple and easy to read (the last three lines).

The code snippet provided includes two functions: `makeMatrix` and `makeNameMap`. The `makeMatrix` function reads a file, splits each line into fields, and constructs a matrix from these fields. The `makeNameMap` function reads the first line of the file and creates a map from the argument string to the index in the matrix.

The `distMatrix` variable is created using `makeMatrix` on the file specified as an argument. Then, `makeNameMap` is called with the same argument. Finally, the distance between the arguments specified as command line arguments is printed.

I highlighted the parts of the code that are particularly noteworthy:

- The `makeMatrix` function is defined to read a file and construct a matrix from the data.
- The `makeNameMap` function reads the first line of the file and creates a mapping from name arguments to matrix indices.
- The `distMatrix` variable is created and printed, using the functions defined.

The code snippet also includes comments explaining the purpose of each function and how they work together to solve the challenge.