

Scientific Programming for Systems Biologists

István Albert

Bioinformatics Consulting Center
Huck Institute for Life Sciences

A bottleneck in science

The inability to perform simple tasks quickly

- Limited amount of time and energy per day, spend it wisely
- Programming is easy - getting started is more difficult
- Goal is not to become a “**programmer**”, but someone who can solve problems **via** programming.

“Nuts and Bolts” Approach

- Lots of examples, cookbook like demonstrations
- This is not “Bioinformatics” - rather “Applied Informatics”
- You might already be familiar with programming concepts

**Programming is about decomposing a problem
to many, very simple, unambiguous steps**

Before your start

- Like any craftsman you need to have the basic tools and basic knowledge
- The **shell** is your workshop, a **text editor** is the work bench
- A basic understanding of how computers work: processor, memory, hard drive, text and binary files, folders (directories),

Software Carpentry: <http://www.swc.scipy.org/>

The Editor

A lot of your time is spent in the editor

Three essential features:

1. ability to execute other programs from the editor
2. line numbering
3. syntax highlighting

Personal preference: **EditPlus** (Windows)
<http://www.editplus.org>

The Shell - text based command

- Cannot be replaced with a graphical interface, despite a concerted effort to “banish” it

- A graphical interface is static, you can only do what others have foreseen you to

- In the shell many simple commands can be chained to form complex statements

- Windows (Command Prompt, Cygwin),
- Mac(builtin),
- Linux (mainly shell based)

Why Python?

There are dozens of other languages

- Can we solve problem X with language Y. **Yes!**
- Does it matter what we use then? **Yes!**
- Languages are optimized for certain goals: fast execution, low memory, particular needs, high order constructs etc.

Python is optimized for Thought.

<http://www.python.org>

Resources for learning Python

Online:

Python Tutorial - <http://docs.python.org/> -> Tutorial

Dive Into Python - <http://www.diveintopython.org/>

Cookbook: <http://aspn.activestate.com/ASPN/Python/Cookbook/>

Books:

Python in a Nutshell

Python Cookbook

Python Essential Reference



The riches of the standard library

<http://docs.python.org/> -> Global Module Index

chunk	functools	stringprep
cmath	gc	struct
cmd	glob (Unix)	subprocess
code	gettextmodule (Mac)	sunau
codecs	gettext	SUNAUDIOCVT (SunOS)
codeop	gettext	sunaudiodev (SunOS)
collections	gettext	symbol
ColorPicker (Mac)	gl (UNIX)	sys
colorsys	glob	syslog (Unix)
commands (Unix)	gopherlib	tabnanny
compileall	grp (Unix)	tarfile
compile	gzip	telnetlib
compiler.ast	hashlib	tempfile
compiler.visitor	heapq	termios (Unix)
ConfigParser	hmac	test
concurrent	hotshot	test.test_support
concurrentlib	hotshot.stats	textwrap
Cookie	htmlentitydefs	thread
cookiecui	htmlib	threading
copy	httplib	time
copy_reg	httpparser	timeit
cPickle	httplib	tk
cProfile	ic (Mac)	Tkinter
crypt (Unix)	icopen (Mac)	token
stringio	imageop	tokenize
csv	imaplib	trace
ctypes	imgfile (UNIX)	traceback
curses	imghdr	tty (Unix)
curses.ascii		turtle (Mac)

External libraries

Numerical - NumPy - <http://numpy.scipy.org/>

Plotting - Matplotlib - <http://matplotlib.sourceforge.net/>

Bioinformatics - BioPython - <http://biopython.org>

Graph Library - NetworkX - <https://networkx.lanl.gov>

MachineLearning - PyML - <http://pyml.sourceforge.net/>

...etc...

Just like learning a foreign language

Need **words** (vocabulary) and need the **rules** (syntax) used to string them together

What's more difficult to learn: words or rules?

Semantics: what does it mean?

Programming very similar: **keywords** -> **syntax** -> **execution**

problem is the computers are "stupid"
they do what you say not what you mean

Keywords and Syntax

Allows us to create standard programming constructs:

- a way to loop over (repeat) a number of statements
- a way to test a condition and branch depending on the result

```
if age > 21:
    print 'Access granted'
else:
    print 'Access denied'
```

```
for age in [1, 2, 3, 4, 5]:
    print age
```

Block structure = Indentation

- Lines indented by the same amount form a block:

```
if age > 21:
    print "Inside block 1"
    print "Still inside 1"
print "Outside"
```

- Nested indentation

```
if age > 21:
    print "Inside block 1"
    if age > 51:
        print "Inside block 2"
        print "Still inside block 2"
        print "Back to block 1"
    print "Outside"
```

Basic constructs

variable assignments

```
name = 'John'
value = 123.4
left = right = 200
```

```
print name, value
```

try:

```
statements
except Exceptions:
statements
```

```
def funcname(parameters):
statements
```

for target **in** iterable:
statements

break # to stop iteration
continue # to force next

if expression:
statements

elif expression:
statements

else:
statements

while expression:
statements

Data Types

- **Numbers:** integers, floats and complex numbers

- **Sequences:**

- **strings** : "John Doe" or 'Jane Doe'
- **tuples** : (1, 99.223, "John")
- **lists** : [1, 2, 3]

- **Mappings:** indexing by another object

- **dictionary** : person={ "name": "Jane Doe", 'age':12}
- **sets** : person = set([1, 2, 3, 4])

programming = juggling with these data types

Creating data types

```
values = [] # empty list
genes = {} # empty dictionary
names = set() # empty set
settings = () # empty tuple ???
anything wrong with an empty tuple?
```

for python lower than 2.4 add -> **from sets import Set as set**

```
values = [ 1, 3, 4 ]
for element in values:
    print element

for supercalifragilisticexpialidocious in values:
    print supercalifragilisticexpialidocious
```

Using Sequences

```
values = [ 1, 200, 300, 400 ]
values.append ( 500 )
print values[0]
print values[2]
print values[10]
print values[-1]
print values[0:2]
print values[1: -1: 2]
```

Mappings - Dictionaries (Hashes)

```
gene = { "name": "FOXP2", "level": 0.22, "pval": 1E-10 }
print gene
print gene["name"]
print gene.keys()
print gene.values()
print gene["FOXP2"]
gene["row"] = 123
print gene["row"]
```

Collections- Sets

```
lo = set( [ "FOXP2", "AN3", "BR4" ] )
hi = set( [ "FOXP2", "AN2", "BR4" ] )

print lo
print 'FOXP2' in lo

hi.add('MIR2')
hi.remove('FOXP2')

print lo - hi
print lo & hi # intersect
print lo | hi # union

for python lower than 2.4 add -> from sets import Set as set
```

Modularizing the code

Functions, modules and packages

```
def greet ( name ):
    return "Hello %s!" % name

print greet( 'John' )
print greet( 'Jane' )
```

```
import greeter

print greeter.greet( 'John' )
print greeter.greet( 'Jane' )
```

OOP - Object oriented programming

```
class Greeter:
    def greet (name):
        print "Hello %s!" % name

greeter = Greeter()
greeter.greet( ' John' )

def greet (name):
    print "Hello %s!" % name

greet( ' John' )
greet( ' Jane' )
```

1. The goal is to create successively bigger building blocks
2. At the same time hide the what we understand
3. Will revisit this in another lecture

Everything is a reference

many language struggle with a duality: *call by value*, or *call by reference* -> leads to a lot of confusion

```
def modify ( values ):
    values.append( 5 )

a = b = [ 1, 2, 3 ]
a.append(4)

print a, b

modify(a)

modify2 = modify
modify2(b)

print a, b
```

Everyday tasks: File Processing

Say we need to process a tab separated GenePix Microarray Result File (GPR)

```
ATF      1.0
28       43
"Type=GenePix Results 1.4"
"DateTime=2003/02/14 12:18:50"
"ImageName=635 nm          532 nm"
"NormalizationFactor:MeanOfRatios=1.34185"
...
"Block" "Column" "Row" "Name" "ID" "X" "Y" "Dia." "F635 Median"
1       1       1     "glucose transporter" "vppeh08" 2500 10070
1       2       1     "VPPEH12 putative protein" "vppeh12" 2770 10070
1       3       1     "tubulin beta-4 chain" "vppeh02" 3040 10070 140
1       4       1     "ribosomal protein S18" "vppeh05" 3300 10060 140
...
3000 such lines
```

First in a set of successively more generic solutions

```
genes = {}
inside = False
for line in file('data.gpr'):
    if inside:
        elems = line.split("\t")
        name = elems[4]
        value = float( elems[8] )
        genes[ name ] = value
    if line.startswith("Block"):
        inside = True

print genes["moz19"] # yuck!

print genes.items()[:10]

# [ ("est_g12g17", 1440.0), ("rpm2-d10", 2054.0), ...]
```

Works but it is ugly -> "special casing"

Using files as iterators

```
reader = file('data.gpr')

for line in reader:
    if line.startswith('Block'):
        break

for line in reader:
    elems = line.split("\t")
    name = elems[4].strip(' ')
    value = float(elems[8])
    genes[name] = value

print genes["moz19"]
print genes.items()[:10]

# [ ('est g12g1t7', 1440.0), ('rpm2-d10', 2054.0), ...]
```

still complicated - fooling around with the quotes, hardcoded values

Use the Docs, Luke!

```
import csv

reader = csv.reader(file('data.gpr'), delimiter="\t")

for elems in reader:
    if elems[0] == "Block":
        gid = elems.index('ID')
        vid = elems.index('F635 Median')
        break

for elems in reader:
    name = elems[gid]
    value = float(elems[vid])
    genes[name] = value

print genes.items()[:10]
```

<http://docs.python.com> -> global module index

File formats

ATF -> second line tells us where the content is, let's use that info

```
ATF 1.0
28 43
"Type=GenePix Results 1.4"
"DateTime=2003/02/14 12:18:50"
"ImageName=635 nm 532 nm"
"NormalizationFactor:MeanOfRatios=1.34185"
...
"Block" "Column" "Row" "Name" "ID" "X" "Y" "Dia." "F635 Median"
1 1 1 "glucose transporter" "vppeh08" 2500 10070
1 2 1 "VPPEH12 putative protein" "vppeh12" 2770 10070
1 3 1 "tubulin beta-4 chain" "vppfa02" 3040 10070 140
1 4 1 "ribosomal protein S18" "vppfa05" 3300 10060 140
...
3000 such lines
```

Doing more with less code

```
import csv

def gpr_loader(fname, delimiter="\t"):
    """
    This function loads a gpr file and returns a list of dicts
    keyed by the header
    """
    fileref = file(fname)

    top = csv.reader(fileref, delimiter=delimiter)
    top.next()
    skip, cols = map(int, top.next()) # ATF format

    [ top.next() for i in range(skip) ]
    bottom = csv.DictReader(fileref, delimiter=delimiter)
    data = [ row for row in bottom ]
    return data

data = gpr_loader('data.gpr')
print len(data)
print data[0]

# { 'B532 Median': '73', 'F532 % Sat.': '0', 'B635 Mean': '85', ... }
```

High level building blocks

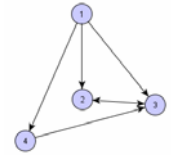
```
import gprutil
data = gprutil.gpr_loader('data.gpr')
print len(data)
print data[0]
# { 'B532 Median': '73', 'F532 % Sat.': '0', 'B635 Mean': '85', ... }
```

- re-factored the code into its own module,
- we're back to a simple "view of the world"
- values are still strings, chain new functionality, don't add into a single step

Lets build a "graph library"

Nested data structure - *dictionary of lists* or *dictionary of dictionaries*

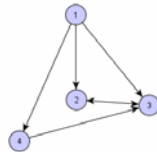
```
graph = { }
for i in range(1,5):
    graph[i] = { }
graph[1][2] = 1
graph[1][3] = 1
graph[1][4] = 1
graph[2][3] = 1
graph[3][2] = 1
graph[4][3] = 1
print graph.keys()
for node in graph:
    print len( graph[node] )
    print graph[node].keys()
print graph[1][4]
```



Refactor the "graph library"

Nested data structure - *dictionary of dictionaries*

```
def add_edge(g, head, tail, dist=1):
    g[head][tail] = dist
graph = { }
for i in range(1,5):
    graph[i] = { }
elems = [ (1,3), (1,4), (2,3), (3,2), (4,3) ]
for head, tail in elems:
    add_edge( g=graph, head=head, tail=tail)
```



Remember this?

```
import random
graph = { }
for i in range(0,10):
    graph[i] = { }
nodes = graph.keys()
for node1 in nodes:
    for node2 in nodes:
        if random.randint(1,6) == 1 and node1 != node2:
            graph[node1][node2] = 1
for node1 in nodes:
    for node2 in nodes[node1+1]:
        if random.randint(1,6):
            graph[node1][node2] = graph[node2][node1] = 1
```

import this

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!