

# Practical Bioinformatics

Mark Voorhies

5/11/2015

Good ideas shamelessly lifted from David Erle and Josh Pollack

# Resources

## Getting “Scientific” Python

- <https://store.enthought.com/#canopy-academic>

# Resources

## Getting “Scientific” Python

- <https://store.enthought.com/#canopy-academic>

## Course website:

- <http://histo.ucsf.edu/BMS270/>

# Resources

## Getting “Scientific” Python

- <https://store.enthought.com/#canopy-academic>

## Course website:

- <http://histo.ucsf.edu/BMS270/>

## Resources on the course website:

- Syllabus
  - Papers and code (for downloading *before* class)
  - Slides and transcripts (available *after* class)
  - (Whiteboard images will be added to slides during class)

# Resources

## Getting “Scientific” Python

- <https://store.enthought.com/#canopy-academic>

## Course website:

- <http://histo.ucsf.edu/BMS270/>

## Resources on the course website:

- Syllabus
  - Papers and code (for downloading *before* class)
  - Slides and transcripts (available *after* class)
  - (Whiteboard images will be added to slides during class)
- On-line textbooks (Dive into Python, Numerical Recipes, ...)
- Programs for this course (Canopy, Cluster3, JavaTreeView, ...)

# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

- Writing standalone scripts.



# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

- Writing standalone scripts.
- Shepherding data between analysis tools.

# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

- Writing standalone scripts.
- Shepherding data between analysis tools.
- Aggregating data from multiple sources.

# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

- Writing standalone scripts.
- Shepherding data between analysis tools.
- Aggregating data from multiple sources.
- Implementing new methods from the literature.

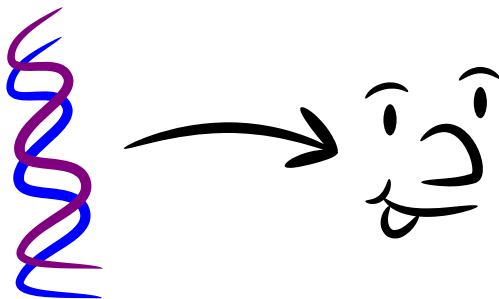
# Goals

At the end of this class, you should have the confidence to take on the day to day tasks of “bioinformatics”.

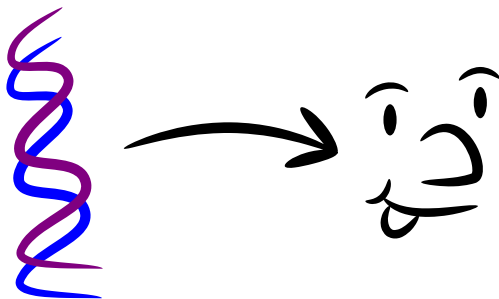
- Writing standalone scripts.
- Shepherding data between analysis tools.
- Aggregating data from multiple sources.
- Implementing new methods from the literature.

This is also good preparation for communicating with computational collaborators.

# Course problems: expression and sequence analysis



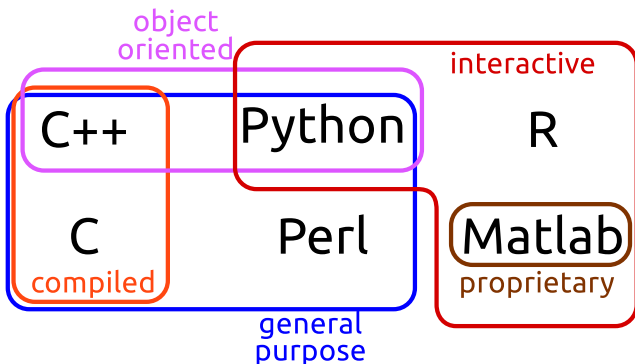
# Course problems: expression and sequence analysis



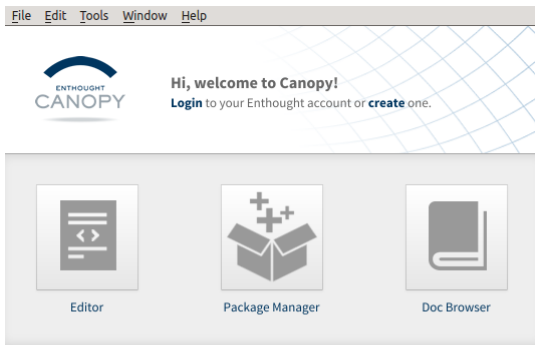
Part 2: Genotype  
(Sequence analysis)

Part 1: Phenotype  
(Expression profiling)

## Course tool: Python



# Python distribution: Enthought Canopy





# Python distribution: Enthought Canopy

The screenshot displays the Enthought Canopy application window. The interface includes a menu bar (Edit, Tools, Window, Help), a search bar, and a sidebar with navigation options: Available Packages, Free Packages, Canopy Packages, Community Packages, Installed Packages (highlighted), Updates 10, and History. The main area shows a list of installed packages with their descriptions and license types.

Package Name	Description	License
jsonpickle 0.4.0	serializing any arbitrary object graph into JSON	CANOPY
kernmagic 0.2.0	adds more magic commands to ipython	FREE
keyring 0.9.2	store and access your passwords safely	CANOPY
libogg 1.3.0		CANOPY
libtheora 1.1.1		CANOPY
libvpx 1.1.0		CANOPY
libxml2 2.7.8	XML parser and toolkit	CANOPY
matplotlib 1.2.0	interactive 2D plotting library	FREE
MKL 10.3	Intel Math Kernel Library (runtime)	FREE
mock 0.7.2		COMMUNITY
nose 1.2.1	extends the test loading and running features of unittest	FREE
numpy 1.6.1	general-purpose array-processing and math library	FREE
pillow 1.1.7		

63 packages installed

# Python distribution: Enthought Canopy

The screenshot shows the Canopy IDE interface. At the top, there is a menu bar with 'Edit', 'Tools', 'Window', and 'Help'. Below the menu bar is a search bar containing 'h5py'. On the left side, there is a sidebar with a tree view showing package categories: 'Available Packages', 'Free Packages', 'Canopy Packages', 'Community Packages', 'Installed Packages', 'Updates 10', and 'History'. The main area displays the details for the 'h5py 2.0.0' package. The package name is 'h5py 2.0.0' and it is described as the 'Python interface to the HDF library'. The version is '2.0.0' and the size is '827.23 kB'. There are buttons for 'Less Info', 'CANOPY', and 'Subscribe'. The description states: 'HDF5 for Python (h5py) is a general-purpose Python interface to the Hierarchical Data Format library, version 5. HDF5 is a versatile, mature scientific software library designed for the fast, flexible storage of enormous amounts of data. From a Python programmer's perspective, HDF5 provides a robust way to store data, organized by name in a tree-like fashion. You can create datasets (arrays on disk) hundreds of gigabytes in size, and perform random-access I/O on desired sections. Datasets are organized in a filesystem-like hierarchy using containers called "groups", and accessed using the traditional POSIX /path/to/resource syntax.' Below the description, the size is listed as '827.23 kB', the version as '2.0.0', and the build as '2'. The dependencies are listed as 'hdf5 1.8.9' and 'numpy 1.6.1'. The MD5 hash is '9571d1d37acd500c734f0094049cd0ea'. At the bottom of the package details, it says 'There are 4 versions of this package' with a 'Show' button. At the very bottom of the window, it says '119 packages available. 1 matches'.

h5py 2.0.0  
Python interface to the HDF library  
Version: 2.0.0  
Size: 827.23 kB

Less Info CANOPY Subscribe

HDF5 for Python (h5py) is a general-purpose Python interface to the Hierarchical Data Format library, version 5. HDF5 is a versatile, mature scientific software library designed for the fast, flexible storage of enormous amounts of data. From a Python programmer's perspective, HDF5 provides a robust way to store data, organized by name in a tree-like fashion. You can create datasets (arrays on disk) hundreds of gigabytes in size, and perform random-access I/O on desired sections. Datasets are organized in a filesystem-like hierarchy using containers called "groups", and accessed using the traditional POSIX /path/to/resource syntax.

Size: 827.23 kB  
Version: 2.0.0  
Build: 2  
Dependencies:  
    hdf5 1.8.9  
    numpy 1.6.1

md5: 9571d1d37acd500c734f0094049cd0ea

There are 4 versions of this package Show

119 packages available. 1 matches

# Python shell: ipython notebook

```
In [5]: np.random.seed(0)

ax = pylab.axes()

x = np.linspace(0, 10, 100)
ax.plot(x, np.sin(x) * np.exp(-0.1 * (x - 5) ** 2), 'b', lw=1, label='damped sine')
ax.plot(x, -np.cos(x) * np.exp(-0.1 * (x - 5) ** 2), 'r', lw=1, label='damped cosine')

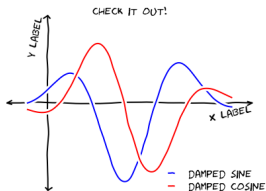
ax.set_title('check it out!')
ax.set_xlabel('x label')
ax.set_ylabel('y label')

ax.legend(loc='lower right')


ax.set_xlim(0, 10)
ax.set_ylim(-1.0, 1.0)

#XXCDisy the axes -- this operates in-place
XXCDisy(ax, xaxis loc=0.0, yaxis loc=1.0,
        xaxis arrow='+-', yaxis_arrow='+-',
        expand_axes=True)
```


Out[5]: <matplotlib.axes.AxesSubplot at 0x2fecbd0>



# Anatomy of a Programming Language

$f(x)$    
functions

# Anatomy of a Programming Language

$f(x)$    
functions

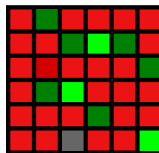


data structures

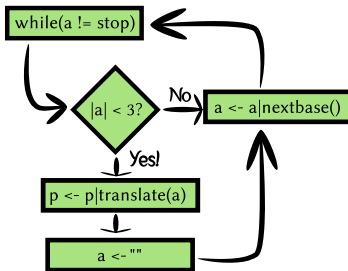
# Anatomy of a Programming Language

$f(x)$  

functions



data structures

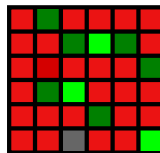


control statements

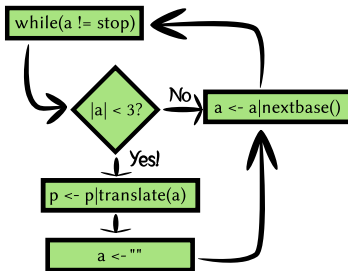
# Anatomy of a Programming Language

$f(x)$  

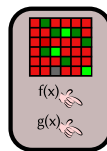
functions



data structures



control statements



objects

## Talking to Python: Nouns

```
# This is a comment
```

```
# This is an int (integer)
```

```
42
```

```
# This is a float (rational number)
```

```
4.2
```

```
# These are all strings (sequences of characters)
```

```
'ATGC'
```

```
"Mendel's Laws"
```

```
""">CAA36839.1 Calmodulin
```

```
MADQLTEEQIAEFKEAFSLFDKDGDTITTKELGTVMRS LGQNPTEAEL
```

```
QDMINEVDADDLPNGTIDFPEFLTMMARKMKD TDSEEEIREAFRVFDK
```

```
DGNGYISAAELRHVMTNLGEKLTDEEVDEMIREADIDGDGQVNYEEFVQ
```

```
MMTAK"""
```



# Python as a Calculator

*# Addition*

1+1

*# Subtraction*

2-3

*# Multiplication*

3\*5

*# Division (gotcha: be sure to use floats)*

5/3.0

*# Exponentiation*

2\*\*3

*# Order of operations*

2\*3-(3+4)\*\*2

# Remembering objects

```
# Use a single = for assignment:
```

```
TLC = "GATACA"
```

```
YFG = "CTATGT"
```

```
MFG = "CTATGT"
```

```
# A name can occur on both sides of an assignment:
```

```
codon_position = 1857
```

```
codon_position = codon_position + 3
```

```
# Short-hand for common updates:
```

```
codon += 3
```

```
weight -= 10
```

```
expression *= 2
```

```
CFU /= 10.0
```

# Displaying values with print

```
# Use print to show the value of an object  
message = "Hello , world"  
print(message)  
# Or several objects:  
print(1,2,3,4)  
# Older versions of Python use a  
# different print syntax  
print "Hello , world"
```

# Comparing objects

*# Use double == for comparison:*

YFG == MFG

*# Other comparison operators:*

*# Not equal:*

TLC != MFG

*# Less than:*

3 < 5

*# Greater than, or equal to:*

7 >= 6

# Making decisions

```
if(YFG == MFG):  
    print "Synonyms!"  
  
if(protein_length < 60):  
    print "Probably too short to fold."  
elif(protein_length > 10000):  
    print "What is this, titin?"  
else:  
    print "Okay, this looks reasonable."
```

# Collections of objects

```
# A list is a mutable sequence of objects
mylist = [1, 3.1415926535, "GATACA", 4, 5]
# Indexing
mylist[0] == 1
mylist[-1] == 5
# Assigning by index
mylist[0] = "ATG"
# Slicing
mylist[1:3] == [3.1415926535, "GATACA"]
mylist[:2] == [1, 3.1415926535]
mylist[3:] == [4, 5]
# Assigning a second name to a list
also_mylist = mylist
# Assigning to a copy of a list
my_other_list = mylist[:]
```

# Repeating yourself: iteration

```
# A for loop iterates through a list one element  
# at a time:
```

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

```
# A while loop iterates for as long as a condition  
# is true:
```

```
population = 1  
while(population < 1e5):  
    print population  
    population *= 2
```

# Verb that noun!

```
return_value = function(parameter, ...)
```

“Python, do *function* to *parameter*”

```
# Built-in functions
```

```
# Generate a list from 0 to n-1
```

```
a = range(5)
```

```
# Sum over an iterable object
```

```
sum(a)
```

```
# Find the length of an object
```

```
len(a)
```



# Verb that noun!

```
return_value = function(parameter, ...)
```

“Python, do *function* to *parameter*”

```
# Importing functions from modules
```

```
import numpy
```

```
numpy.sqrt(9)
```

```
import matplotlib.pyplot as plt
```

```
fig = plt.figure()
```

```
plt.plot([1,2,3,4,5],  
         [0,1,0,1,0])
```

```
from IPython.core.display import display
```

```
display(fig)
```

# New verbs

```
def function(parameter1, parameter2):  
    """Do this!"""  
    # Code to do this  
    return return_value
```

# Summary

- Python is a general purpose programming language.

# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).

# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).
- We can define complex behaviors through control statements like “for”, “while”, and “if” .

# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).
- We can define complex behaviors through control statements like "for", "while", and "if".
- We can use an interactive Python session to experiment with new ideas and to explore data.

# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).
- We can define complex behaviors through control statements like “for”, “while”, and “if” .
- We can use an interactive Python session to experiment with new ideas and to explore data.
- Saving interactive sessions is a good way to document our computer “experiments” .

# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).
- We can define complex behaviors through control statements like "for", "while", and "if".
- We can use an interactive Python session to experiment with new ideas and to explore data.
- Saving interactive sessions is a good way to document our computer "experiments".
- Likewise, we can use modules and scripts to document our computer "protocols".



# Summary

- Python is a general purpose programming language.
- We can extend Python's built-in functions by defining our own functions (or by importing third party modules).
- We can define complex behaviors through control statements like "for", "while", and "if".
- We can use an interactive Python session to experiment with new ideas and to explore data.
- Saving interactive sessions is a good way to document our computer "experiments".
- Likewise, we can use modules and scripts to document our computer "protocols".
- Most of these statements are applicable to any programming language (Perl, R, Bash, Java, C/C++, FORTRAN, ...)

# Homework

- E-mail Mark your python sessions (.ipynb files) after class
- E-mail Mark any homework code/results before tomorrow's class

# Homework

- E-mail Mark your python sessions (.ipynb files) after class
- E-mail Mark any homework code/results before tomorrow's class
- It is fine to work together and to consult books, the web, etc. (but let me know if you do)
- It is fine to e-mail Mark questions
- Don't blindly copy-paste other people's code (you won't learn)

# Homework

- E-mail Mark your python sessions (.ipynb files) after class
- E-mail Mark any homework code/results before tomorrow's class
- It is fine to work together and to consult books, the web, etc. (but let me know if you do)
- It is fine to e-mail Mark questions
- Don't blindly copy-paste other people's code (you won't learn)
- If you get stuck, try working things out on paper first.
- Do as much as you can in about 2 hours

# Homework

- E-mail Mark your python sessions (.ipynb files) after class
- E-mail Mark any homework code/results before tomorrow's class
- It is fine to work together and to consult books, the web, etc. (but let me know if you do)
- It is fine to e-mail Mark questions
- Don't blindly copy-paste other people's code (you won't learn)
- If you get stuck, try working things out on paper first.
- Do as much as you can in about 2 hours (unless you're really enjoying yourself)

# Homework: Make your own Fun

Write functions for these calculations, and test them on random data:

- 1 Mean:

$$\bar{x} = \frac{\sum_i^N x_i}{N}$$

- 2 Standard deviation:

$$\sigma_x = \sqrt{\frac{\sum_i^N (x_i - \bar{x})^2}{N - 1}}$$

- 3 Correlation coefficient (Pearson's r):

$$r(x, y) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2} \sqrt{\sum_i (y_i - \bar{y})^2}}$$