Practical Bioinformatics

Mark Voorhies

4/18/2011

Resources

Router:

- SSID: BMS270
- password: deoxyribose

Resources

Router:

- SSID: BMS270
- password: deoxyribose

Getting Python

- http://www.python.org/download

Resources

Router:

- SSID: BMS270
- password: deoxyribose

Getting Python

- http://www.python.org/download

Course website:

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

Resources

Router:

- SSID: BMS270
- password: deoxyribose

Getting Python

- http://www.python.org/download

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)
- On-line textbooks (Safari Bookshelf, Numerical Recipes, ...)
- Programs for this course (Python, Cluster3, JavaTreeView, ...)

Goals

At the end of this class, you should have the con dence to take on the day to day tasks of \bioinformatics".

Goals

At the end of this class, you should have the con dence to take on the day to day tasks of \bioinformatics".

- Writing standalone scripts.

Goals

At the end of this class, you should have the con dence to take on the day to day tasks of \bioinformatics".

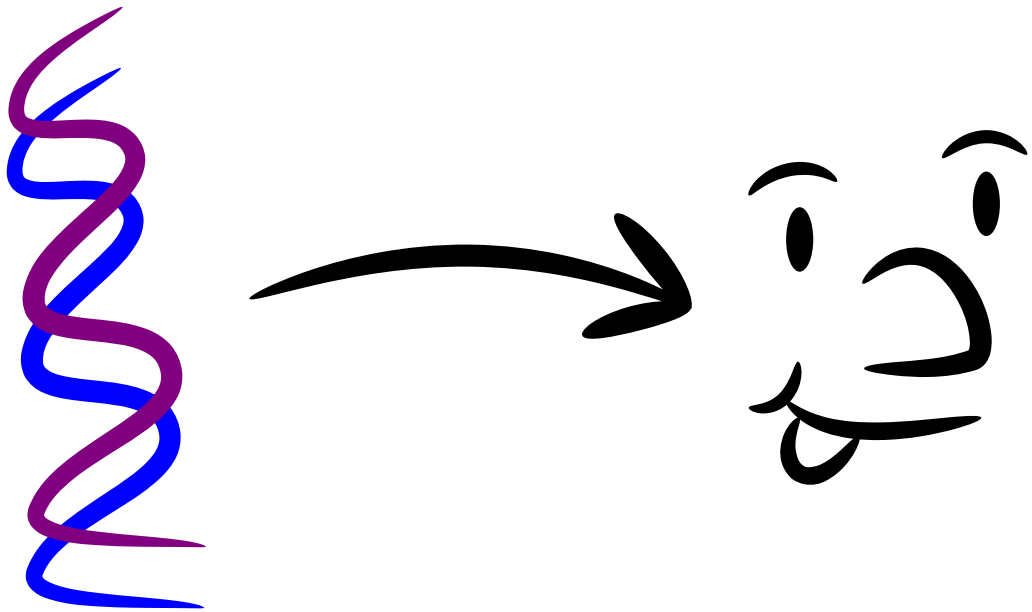
- Writing standalone scripts.
- Shepherding data between analysis tools.
- Aggregating data from multiple sources.

Goals

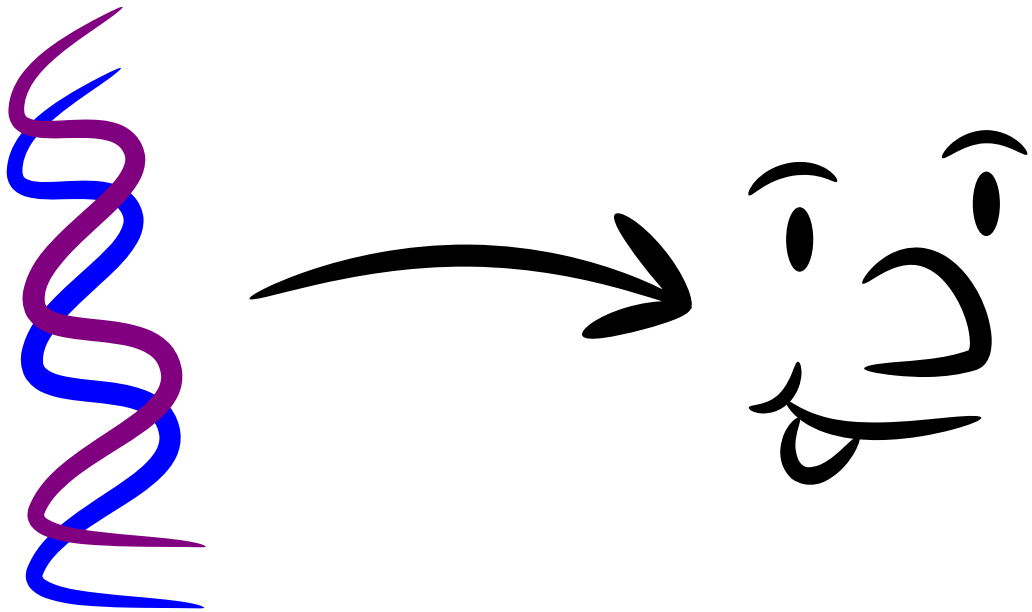
At the end of this class, you should have the con dence to take on the day to day tasks of \bioinformatics".

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

Course problems: microarray and sequence analysis



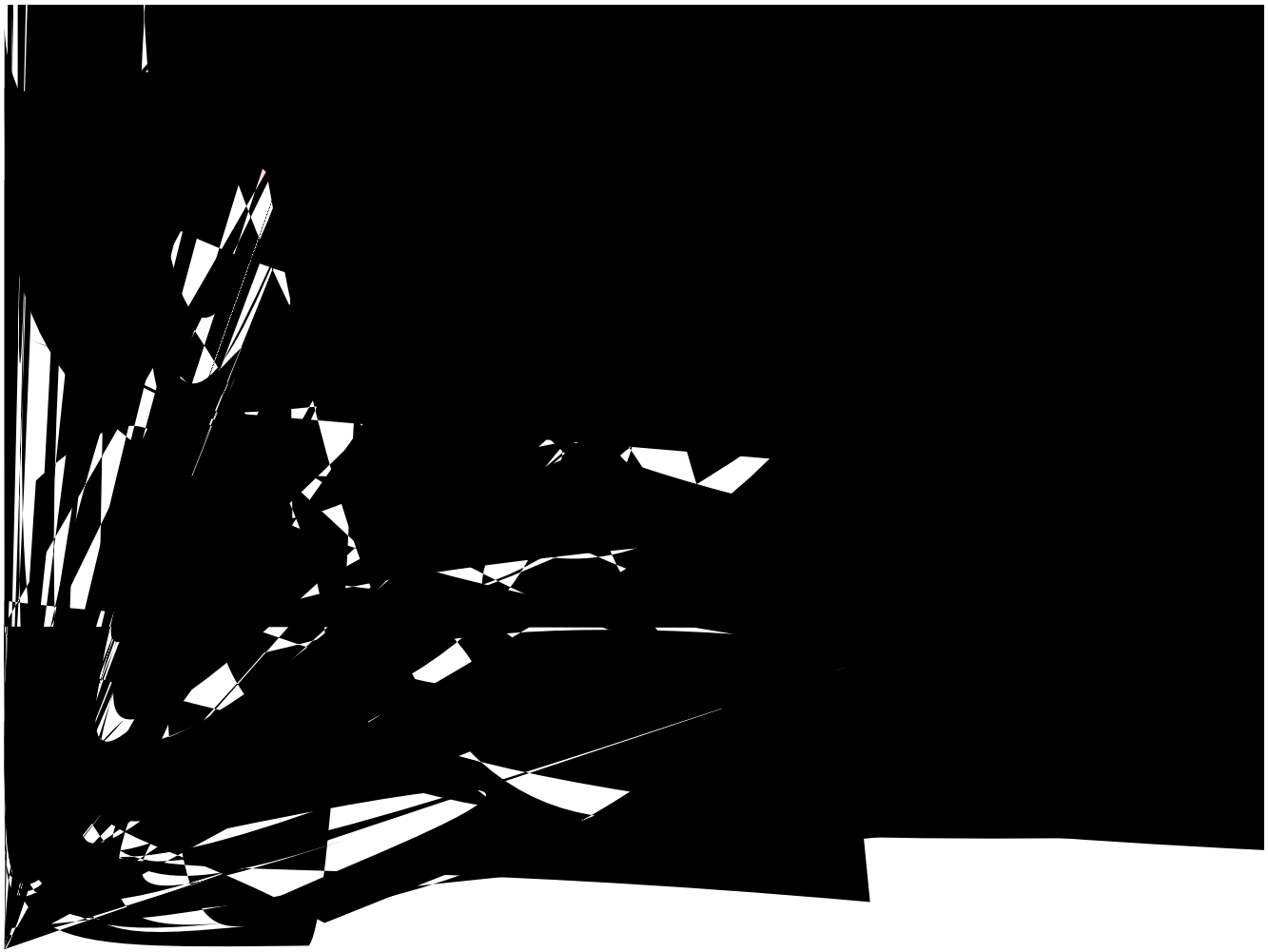
Course problems: microarray and sequence analysis



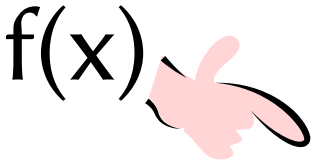
Part 2: Genotype
(Sequence analysis)

Part 1: Phenotype
(Expression pro ling)

Anatomy of a Programming Language

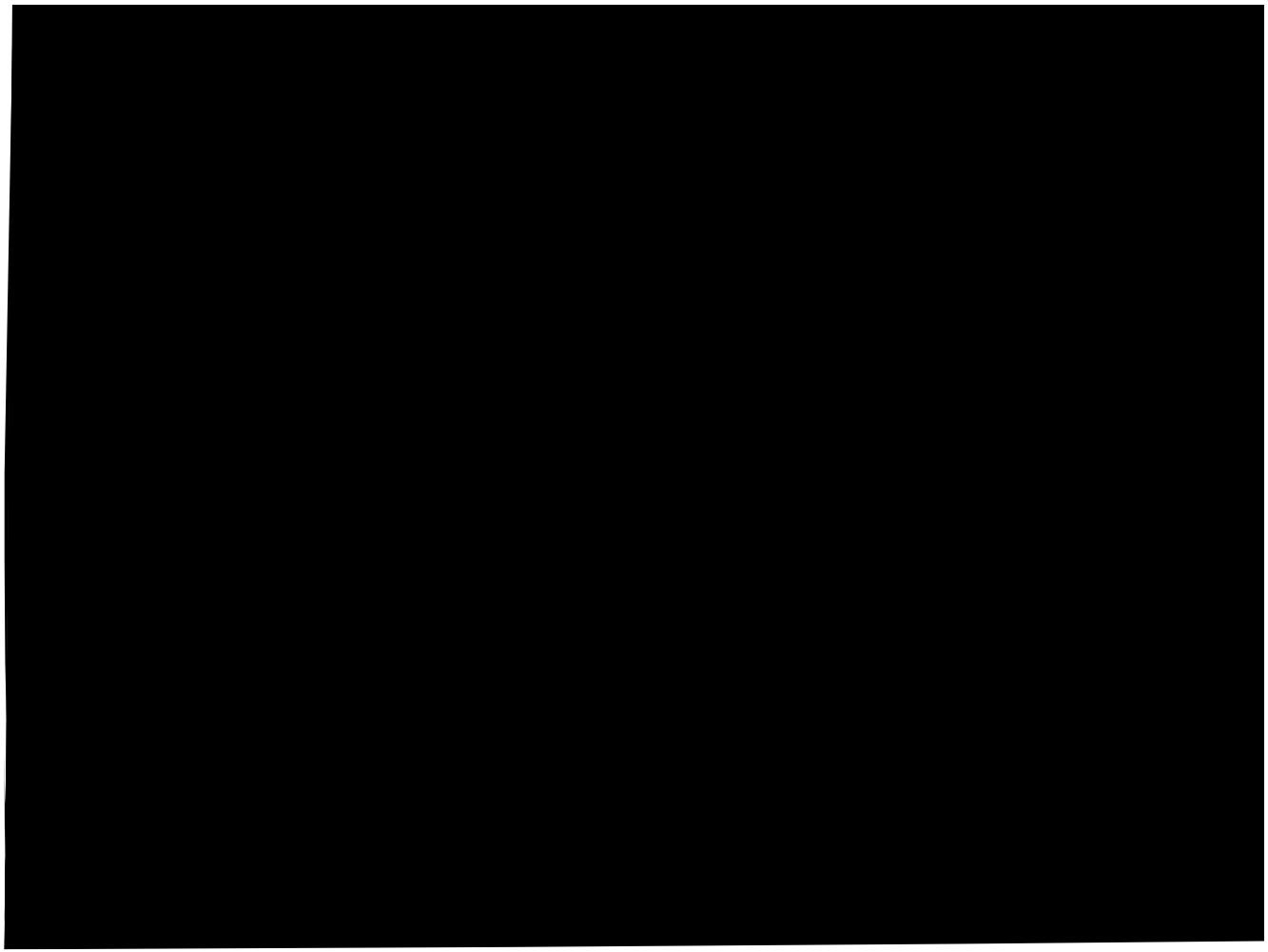


Anatomy of a Programming Language

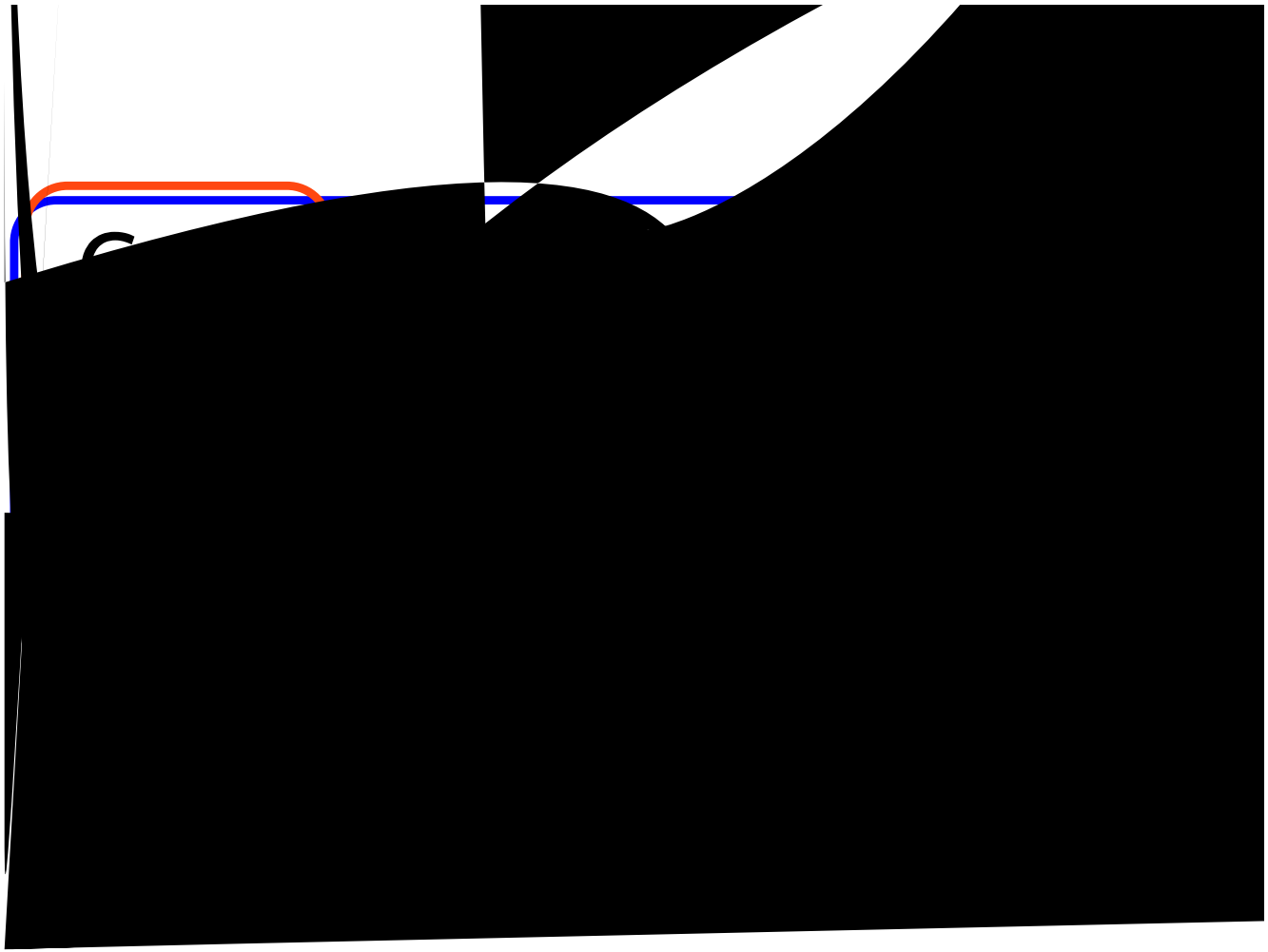


Anatomy of a Programming Language

Anatomy of a Programming Language



Course tool: Python



Two ways to start Python on OS X

- Using IDLE (like Windows)
  1. Open a terminal window
  2. python -m idlelib.idle
- Using command-line Python and Emacs (more stable)
  1. Open a terminal window
  2. python
  3. Open a second terminal window (or a new tab)
  4. emacs

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
MADQLTEEQIAEFKEAFSLFDKDGDGTITTKELGTVMRSLGQNPTEAEL
QDMINEVDADDLPGNGTIDFPEFLTMMARKMKDTDSEEEIREAFRVFDK
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
```

Python as a Calculator

1. Calculate the T_m of a QuickChange mutagenesis primer with length 25bp ($L = 25$), 13 GC bases ($n_{GC} = 13$), and 2 mismatches to the template ($n_{MM} = 2$) using the formula from Stratagene:

$$T_m = 81.5 + \frac{41n_{GC} - 100n_{MM} - 675}{L} \quad (1)$$

2. Calculate the molarity of a 70mer oligonucleotide with $A_{260} = .03$ using the formula from Maniatis:

$$C = \frac{.02A_{260}}{330L} \quad (2)$$

3. Find the original concentration assuming 10μL of oligo diluted into 990μL of solvent for the measurement.

Saving and comparing 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
```

Checking 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
```

Saving and 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
```

Saving and comparing objects

```
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[:]
```

Summary statistics

Given a list of $\log_2$ expression ratios:
$x = [1.8, 2.0, 1.7, 1.9, 2.3, 1.6, 2.2, 1.8, 1.9, 4.0, 1.7]$

1. Print the corresponding expression ratio values
2. Calculate the mean (average) $\log_2$ ratio:

$$x = \frac{\sum_{i=1}^{N} x_i}{N}$$

3. Calculate the mean expression ratio
4. In what situations do either of these mean values capture useful information about our measurements?

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)
```

Summary statistics

Given a list of $\log_2$ expression ratios:
$x = [1.8, 2.0, 1.7, 1.9, 2.3, 1.6, 2.2, 1.8, 1.9, 4.0, 1.7]$

1. Calculate the mean (average) $\log_2$ ratio:

$$x = \frac{\sum_{i=1}^{N} x_i}{N}$$

using functions to simplify your calculation.

Fun with logarithms

In log space, multiplication and division become addition and subtraction:

$$\begin{aligned} \log(xy) &= \log(x) + \log(y) \\ \log(x/y) &= \log(x) - \log(y) \end{aligned}$$

Fun with logarithms

In log space, multiplication and division become addition and subtraction:

$$\begin{aligned} \log(xy) &= \log(x) + \log(y) \\ \log(x/y) &= \log(x) - \log(y) \end{aligned}$$

Therefore, exponentiation becomes multiplication:

$$\log(x^y) = y \log(x)$$

Fun with logarithms

In log space, multiplication and division become addition and subtraction:

$$\begin{aligned} \log(xy) &= \log(x) + \log(y) \\ \log(x/y) &= \log(x) - \log(y) \end{aligned}$$

Therefore, exponentiation becomes multiplication:

$$\log(x^y) = y \log(x)$$

Also, we can change of the base of a logarithm like so:

$$\log_A(x) = \log(x)/\log(A)$$

Verb that noun!

return_value = function(parameter, ...)
\Python, do *function* to *parameter*"

```
# Importing functions from modules
import math
math.sqrt(9)
math.log(8)/math.log(2)

from math import log
log(16)/log(2)
```

Summary statistics

Given a list of expression ratios:

$r = [4.00, 4.59, 3.73, 4.29, 5.66, 3.48, 5.28, 4.00, 4.29, 18.38, 3.73]$

1. Write a for loop to convert the list to $\log_2$ ratios
2. How can you do this conversion without destroying the original list?

Short-hand for converting lists

```
from math import log
log2 = log(2)
logratios = [log(i)/log2 for i in ratios]
```

New verbs

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

Setting IDLE's working directory

OS X

- Open a terminal
- cd path/to/working/directory
- python -m idlelib.idle

Windows

-

Emacs cheat sheet

C = control, M = meta

C-x C-c	Quit
C-g	Break out of current command
C-_	Undo last edit
C-x C-s	Save
C-x C-w	Save as
C-x C-f	Open/create file (in new buffer)
C-x b	Switch buffers
C-x k	Kill buffer (close file)
C-k	Kill (cut) line
C-w	Kill (cut) selection
C-y	Yank (paste)

Loading and re-loading your functions

```
# Use import the first time you load a module
#   (And keep using import until it loads
#    successfully)
import my_module

my_module.my_function(42)

# Once a module has been loaded, use reload to
#   force python to read your new code
reload(my_module)
```

Make your own Fun

Write functions for these calculations:

1. Mean:

$$x = \frac{\sum_i^N x_i}{N} \tag{3}$$

2. Standard deviation:

$$\sigma_x = \sqrt{\frac{\sum_i^N (x_i - x)^2}{N - 1}} \tag{4}$$

3. Correlation coe cient (Pearson's r):

$$r(x, y) = \frac{\sum_i (x_i - x)(y_i - y)}{\sqrt{\sum_i (x_i - x)^2} \sqrt{\sum_i (y_i - y)^2}} \tag{5}$$

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 de ning 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 de ning our own functions (or by importing third party modules).
- We can de ne 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 de ning our own functions (or by importing third party modules).
- We can de ne 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 de ning our own functions (or by importing third party modules).
- We can de ne 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 de ning our own functions (or by importing third party modules).
- We can de ne 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 de ning our own functions (or by importing third party modules).
- We can de ne 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

1. Practice writing your own functions and importing them into Python.
2. Read Sections 4.1{4.4 of Learning Python, 3rd edition.
   - (If you are feeling ambitious, read 8.1-8.2, 12, and 13).
3. Remember to come to class tomorrow!