Lab 1: Introduction to Python Programming
A few preliminary words...
Overview

• Schedule
• Logistics
• Getting Started
• Into to Unix
• Intro to Python
• Assignment 1
Getting the most out of this course

1. Start the homework EARLY
2. Collaborate
3. Use your resources – tutors, TAs, professors, labmates, Blackboard discussions, and most of all, the internet.
4. Think big
Logistics

- Register for 4 credits
- Labs are a continuation of the concepts learned from lectures
- Lab material is generally not tested on exams
- Course website: http://genetics.wustl.edu/bio5488/
- Bring your laptop to every lab
Where to get help
(a.k.a. how to maintain your sanity)

• Come to office hours
  • Mondays after class (11:30-12:30 pm) in the 4th floor classroom 4515 McKinley and by appointment
  • E-mail if you will cannot come in the first 30 minutes

• Come to tutoring sessions
  • Tuesdays 5:30pm-7pm in 6001B Couch Biomedical Research Building

• Email bio5488wustl@gmail.com

• Work in groups

• Google, google, google
Assignments

• Assignments are posted on the course website Wednesdays at 10am
• Assignments are due the following Friday at 10am

• Assignment format
  • Given a bioinformatics problem
  • Write/complete a Python script
  • Analyze data with your script
  • Answer biological questions about your results

• Turn in format
  • More on this in a bit 😊
Assignment policies

• See the Course Information → Assignment policies document in the course website

• There are 13 assignments
  • You must turn in all assignments
  • All assignments are weighted equally

• Late policy
  • 25% penalty for turning in assignment 1 day late
  • Assignments that are > 1 day late will given a 0
  • Email us at least 24 hours prior to deadline to request an extension

• Collaboration
  • Group work is encouraged, but plagiarism is unacceptable
  • Try to “Google it” first
  • Cite your sources

• Read the assignment before coming to lab
Grading

• Each assignment is out of 10 points

• Graded on
  • Does the code work?
    • It doesn’t have to be the “fastest” or “most efficient” to get full credit
    • If doesn’t work, describe where you had problems
    • Is the code well commented and readable? (more on commenting later 😊)
  • Are the answers correct?

• Grades will be returned in a file called grades.txt on the class server
  • Only you and the TAs will be able to read this file
Getting started
How do I access the server?

• The way we are using here is command-line interfaces (CLI)

• A terminal emulator is a program that allows you to interact with the shell through a CLI
  • There are many different terminal programs that vary across OSs
  • We’ll be using PuTTY (Windows) and Terminal (Mac)
How to log onto the remote computer

Live Demo
How to log onto the remote computer (Windows users)

1. Launch Putty
2. In the host name field, enter genomic.wustl.edu
3. In the port field, enter 22
4. Enter a session nickname, e.g., bio5488
5. Click Save
6. Click Open
How to log onto the remote computer

(Mac users)

1. Open Terminal (found in /Applications/Utilities)
How to log onto the remote computer (Mac users)

2. SSH to the remote computer. Type:

   \texttt{ssh <username>@genomic.wustl.edu}

where \texttt{<username>} is replaced with your username

3. A security message may be printed. Type \texttt{yes} and hit enter.
How to log onto the remote computer (Mac users)

4. Enter your password - *it will not show that you are typing!* Hit enter.
A couple of notes

• When you log onto the class server you will be located in YOUR home directory.

• Every command that you run after logging onto a remote computer will be run on that computer.
Exercise: changing your password (passwd)

• To change your password, type the command
  
  $ passwd

• This will launch the interactive password changer
  • It will ask you for your current password and your new password twice
  • When typing your password, *it will not show that you are typing!*

• Example
  
  $ passwd
  Changing password for nrockweiler.
  (current) UNIX password:
  Enter new UNIX password:
  Retype new UNIX password:
  passwd: password updated successfully
Sublime Text

• Sublime Text is a **text editor** for writing and editing scripts
• We’ll use Sublime to edit both local and remote files
• Installation: [https://www.sublimetext.com/3](https://www.sublimetext.com/3)
• Documentation: [http://www.sublimetext.com/support](http://www.sublimetext.com/support)
Cyberduck

- Cyberduck is a secure file transfer client and will allow you to transfer files from your local computer to a remote computer.
Exercise: setting up Cyberduck

• Create a bookmark
  • Launch the Cyberduck application
  • Click Bookmark → New Bookmark
  • Select SFTP (SSH File Transfer Protocol) from the drop down menu
  • Enter a nickname for the bookmark, e.g., bio5488
  • Enter genomic.wustl.edu as the server name
  • Click the X

• Set the default text editor
  • Click Edit → Preferences → Editor
  • Select sublime text from the drop down menu. (You may need browse your computer for the editor)
  • Check Always use this application
  • Restart Cyberduck
Exercise: transferring files with Cyberduck

• To *download* a file to your local computer
  • Drag and drop a file from Cyberduck to your Finder/File Explorer window
  • Or, double-click

• To *upload* a file to the remote computer
  • Drag and drop a file from Finder/File Explorer to Cyberduck
Exercise: editing remote files with Sublime Text and Cyberduck

• New files
  • Click File → New file
  • Enter a filename
  • Click edit
  • Sublime Text should now launch
  • Add some text to the file
  • Click File → Save or ctrl+S

• Existing files
  • Select the file by clicking the filename 1X
  • Click the Edit button in the navigation bar
  • Edit the file
  • Click File → Save or ctrl+S
Attention about using Cyberduck:

• When clicking on:
  
  o Make sure you see this

• When saving the file, make sure you see the following to make sure the upload is complete before you close the editor

• Before closing the editor, check the time stamp of file
FileZilla

• FileZilla is an alternative approach for Cyberduck
• Can be downloaded for free here: https://filezilla-project.org/
Follow the instructions.

Finally, we should see this.
Basic Unix
A few (more) preliminary words...

A lot of Unix skills revolve around moving around the file system

- This concept is similar to using Apple Finder or the Windows File Explorer GUIs, only this time, we can’t use a mouse or see any fancy graphics 😞
- Be patient, the familiarity will come eventually
Determining where you are (\textit{pwd})

• If you get lost in the file system, you can determine where you are by typing:
  \begin{verbatim}
  $ \textit{pwd}
  /home/nrockweiler
  \end{verbatim}

• \textit{pwd} stands for \textit{p}rint \textit{w}orking \textit{d}irectory

• \textit{pwd} prints the full path of the \textit{current working directory}
Listing directory contents

(\texttt{ls})

\begin{itemize}
  \item To list the contents of a directory:
    \begin{verbatim}
    \$ \texttt{ls}
    \end{verbatim}

    \texttt{assignment1 foo}
  \item \texttt{ls} stands for \texttt{list directory contents}
\end{itemize}
Bio5488 command convention

• We highly recommend that you type all of the command/code yourself rather than copy and pasting

• Here's an example of a command line "snippet“

Template:

$ type_me_exactly <modify_me>

Example:

$ ls <assignment>

README.txt
Changing directories (cd)

• To change to different directory
  
  $ cd <directory_name>

  where

  <directory_name> = the path you want to move to

  • A path is a location in the file system

• cd stands for change directory

• To get back to your home directory

  $ cd ~

• ~ is shorthand for your home directory
Changing directories (cont.)

• To move *one* directory above the current directory
  
  $ cd ../$

• To move *two* directories above the current directory
  
  $ cd ..../..

• You can string as many ../ as you need to
Making directories (mkdir)

- To make a directory
  
  ```
  $ mkdir <new_directory_name>
  ```

  where

  `<new_directory_name>` = name of the directory to create

- `mkdir` stands for **m**ake **d**irectory

- Do not use spaces or “/” in directory or file names
Exercise: create some directories

Try to create this directory structure:

Hints

• Use `pwd` to determine where you are in the directory structure

• Use `cd` to navigate through the directory structure.

• Use `mkdir` to create new directories
Copying things

(cp)

• To create a copy of a file

$ cp -i <filename> <copy_of_filename>

where

  <filename> = file you want to copy
  <copy_of_filename> = name of copied file

The -i flag is a safety feature to make sure you do not overwrite a file that already exists

• To create a copy of a directory

$ cp -r <directory> <copy_of_directory>

where

  <directory> = directory you want to copy
  <copy_of_directory> = name of copied directory

The -r flag is required to copy all of the directory’s files and subdirectories
Copying things (cont.)

**(cp)**

- **cp** stands for *copy* files/directories
- To create a copy of file *and keep the name the same*
  
  
  $ cp -i <filename> .

  where

  `<filename>` = file you want to copy

  - The shortcut is the same for directories, just remember to include the `-r` flag
Exercise: copying things

Copy /home/assignments/assignment1/README.txt to your work directory. Keep the name the same.
Renaming/moving things

(mv)

• To rename/move a file/directory

$ mv -i <original_filename> <new_filename>

where

<original_filename> = name of file/dir you want to rename
<new_filename> = name you want to rename it to

• mv stands for move files/directories
Printing contents of files (cat)

• To print a file

$ cat <filename>

where

<filename> = name of file you want to print

• cat stands for concatenate file and print to the screen

• Other useful commands for printing parts of files:
  • more
  • less
  • head
  • tail
Exercise: printing contents of files

Print the contents of your README.txt

Experiment with using different commands, e.g., cat, head, and tail. How do the commands differ?
Deleting Things

$ \texttt{rm <file\_to\_delete>}$

where

$\langle\text{file\_to\_delete}\rangle =$ name of the file you want to delete

$\texttt{rm -r -i <directory\_to\_delete>}$

where

$\langle\text{directory\_to\_delete}\rangle =$ name of the directory you want to delete

$\texttt{rm}$ stands for $\textit{r}e\textit{move}$ files/directories

**IMPORTANT:** there is no recycle bin/trash folder on Unix!! Once you delete something, it is gone forever. Be very careful when you use $\texttt{rm}$!!

**TIP:** Check that you’re going to delete the correct files by first testing with ‘ls’ and then committing to ‘rm’
Exercise: deleting things

Delete the test directory that you created in a previous exercise.
Saving output to files

• *Save* the output to a file
  
  $ \langle \text{cmd} \rangle \ > \ <\text{output\_file}>$

  where

  \[
  \langle \text{cmd} \rangle = \text{command}
  \]

  \[
  <\text{output\_file}> = \text{name of output file}
  \]

  • **WARNING:** this will overwrite the output file if it already exists!

• *Append* the output to the end of a file

  
  $ \langle \text{cmd} \rangle \ >> \ <\text{output\_file}>$

  There are 2 “>”
Learning more about a command

(\texttt{man})

- To view a command’s documentation
  
  \$ \texttt{man} \text{<cmd>}
  
  where
  
  \texttt{<cmd>} = \text{command}

- \texttt{man} stands for \textit{manual page}

- Use the \texttt{\uparrow} and \texttt{\downarrow} arrow keys to scroll through the manual page

- Type “q” to exit the manual page
Exercise: reading documentation

Determine what the following command does

$ cal -3
Getting yourself out of trouble

• Abort a command
  
  ctrl + C

• Temporarily stop a command
  
  ctrl + Z
Unix commands cheatsheet--your new bestie

<table>
<thead>
<tr>
<th>File Commands</th>
<th>File Permissions</th>
<th>Process Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>ls - directory listing</td>
<td>chmod octal file - change the permissions of file to octal, which can be found separately for user, group, and world by adding:</td>
<td>ps - display your currently active processes</td>
</tr>
<tr>
<td>ls -al - formatted listing with hidden files</td>
<td>● 4 - read (r)</td>
<td>top - display all running processes</td>
</tr>
<tr>
<td>cd dir - change directory to dir</td>
<td>● 2 - write (w)</td>
<td>kill pid - kill process id pid</td>
</tr>
<tr>
<td>cd - change to home</td>
<td>● 1 - execute (x)</td>
<td>killall proc - kill all processes named proc *</td>
</tr>
<tr>
<td>pwd - show current directory</td>
<td>Examples:</td>
<td>bg - lists stopped or background jobs; resume a stopped job in the background</td>
</tr>
<tr>
<td>mkdir dir - create a directory dir</td>
<td>chmod 777 - read, write, execute for all</td>
<td>fg - brings the most recent job to foreground</td>
</tr>
<tr>
<td>rm file - delete file</td>
<td>chmod 755 - rwx for owner, rx for group and world</td>
<td>fg n - brings job n to the foreground</td>
</tr>
<tr>
<td>rm -r dir - delete directory dir</td>
<td>For more options, see man chmod.</td>
<td></td>
</tr>
</tbody>
</table>
Python basics
What is Python?

- Python is a widely used programming language
- Language started in 1989 by Guido van Rossum
- Free, open-source software with community-based development
- Trivia: Python is named after the BBC show “Monty Python’s Flying Circus” and has nothing to do with reptiles

Which Python?

- There are 2 widely used versions of Python: Python2.7 and Python3.x
- We’ll use Python3
- Many help forums still refer to Python2, so make sure you’re aware which version is being referenced
Interacting with Python

• Two Main Ways:
  o Interactive mode
    ▪ Start Interactive mode via
      `python3`
  o Normal mode
    ▪ Execute a script via
      `python3 <script name>`

• Live Demo
Variables

• The most basic component of any programming language are "things," also called **variables**
• A variable has a name and an associated value
• The most common types of variables in Python are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>A whole number</td>
<td>x = 10</td>
</tr>
<tr>
<td>Floats</td>
<td>A real number</td>
<td>x = 5.6</td>
</tr>
<tr>
<td>Strings</td>
<td>Text (1 or more <strong>characters</strong>)</td>
<td>x = “Genomics”</td>
</tr>
<tr>
<td>Booleans</td>
<td>A binary outcome: true or false</td>
<td>x = True</td>
</tr>
</tbody>
</table>

You can use single quotes or double quotes.
Variables (cont.)

• To save a variable, use =

  >>> x = 2

  The value of the variable
  The name of the variable

• To determine what type of variable, use the type function

  >>> type(x)
  <class 'int'>

• IMPORTANT: the variable name must be on the left hand side of the =

  >>> x = 2 😄
  >>> 2 = x 😞
Variable naming (best) practices

• Must start with a letter
• Can contain letters, numbers, and underscores ➔ no spaces!
• Python is case-sensitive: x ≠ X
• Variable names should be descriptive and have reasonable length
• Use ALL CAPS for constants, e.g., PI
• Do not use names already reserved for other purposes (min, max, int)

Want to learn more tips? Check out http://www.makinggoodsoftware.com/2009/05/04/71-tips-for-naming-variables/
Exercise: defining variables

• Create the following variables for
  • Your favorite gene name
  • The expression level of a gene
  • The number of upregulated genes
  • Whether the \textit{HOXA1} gene was differentially expressed

• What is the type for each variable?

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>A whole number</td>
<td>x = 10</td>
</tr>
<tr>
<td>Floats</td>
<td>A real number</td>
<td>x = 5.6</td>
</tr>
<tr>
<td>Strings</td>
<td>Text (1 or more \textit{characters})</td>
<td>x = “Genomics”</td>
</tr>
<tr>
<td>Booleans</td>
<td>A binary outcome: true or false</td>
<td>x = True</td>
</tr>
</tbody>
</table>

Cheatsheet
Collections of things

- Python has several types of data collection structures
  - **Lists** (similar to arrays)
  - **Tuples**
  - **Dictionaries**
Lists: what are they?

- Lists hold a collection of things in a **specified order**
  - The things do not have to be the same type
- Many methods can be used to manipulate lists.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a list</td>
<td><code>&lt;list_name&gt; = [&lt;item1&gt;, &lt;item2&gt;]</code></td>
<td><code>genes = ['SUCLA2', 'SDHD']</code></td>
</tr>
<tr>
<td>Index a list</td>
<td><code>&lt;listname&gt;[&lt;position&gt;]</code></td>
<td><code>genes[1]</code></td>
</tr>
</tbody>
</table>
Lists: where can I learn more?

• Python.org tutorial: [https://docs.python.org/3.4/tutorial/datastructures.html#more-on-lists](https://docs.python.org/3.4/tutorial/datastructures.html#more-on-lists)

• Python.org documentation: [https://docs.python.org/3.4/library/stdtypes.html#list](https://docs.python.org/3.4/library/stdtypes.html#list)
Doing stuff to variables

• There are 3 common tools for manipulating variables
  • Operators
  • Functions
  • Methods
Operators

• Operators are a special type of function:
  • Operators are symbols that perform some mathematical or logical operation

• Basic mathematical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>&gt;&gt;&gt; 2 + 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>&gt;&gt;&gt; 2 - 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>&gt;&gt;&gt; 2 * 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>&gt;&gt;&gt; 2 / 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.66666666666666666666</td>
</tr>
</tbody>
</table>
You can also use operators on strings!

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Combine strings together</td>
<td>&gt;&gt;&gt;&gt; 'Bio' + '5488'</td>
</tr>
</tbody>
</table>

Is it a bird? Is it a plane? No it's a string!

Strings and ints cannot be combined.
Relational operators

- Relational operators compare 2 things
- Return a boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>2 &lt; 3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>2 &lt;= 3</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>2 &gt; 3</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>2 &gt;= 3</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>2 == 3</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>2 != 3</td>
</tr>
</tbody>
</table>

== is used to test for equality
= is used to assign a value to a variable
Logical operators

• Perform a logical function on 2 things
• Return a boolean

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Return True if <em>both</em> arguments are true</td>
<td>&gt;&gt;&gt;&gt; True and True</td>
</tr>
</tbody>
</table>
Functions: what are they?

• Why are functions useful?
  • Allow you to reuse the same code
    • Programmers are lazy!
  • A block of reusable code used to perform a specific task

• Similar to mathematical functions, e.g., \( f(x) = x^2 \)
• 2 types:

**Built-in**
  Function prewritten for you
  - `print`: print something to the terminal
  - `float`: convert something to a floating point #

**User-defined**
  You create your own functions
Functions: how can I call a function?

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call a function that takes no arguments</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;function_name&gt;()</code></td>
<td><code>sys.exit()</code></td>
<td></td>
</tr>
</tbody>
</table>
Python functions: where can I learn more?

• Python.org tutorial
  • User-defined functions: https://docs.python.org/3/tutorial/controlflow.html#defining-functions

• Python.org documentation
  • Built-in functions: https://docs.python.org/3/library/functions.html
Methods: what are they?

• First a preamble...
  • Methods are a close cousin of functions
  • For this class we’ll treat them as basically the same
  • The syntax for calling a method is different than for a function
  • If you want to learn about the differences, google object oriented programming (OOP)

• Why are functions methods useful?
  • Allow you to reuse the same code
# String methods

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| `<str>`.upper() | • Returns the string with all letters uppercased                          | ```
|x = "Genomics"
>>> x.upper()
``` |
Making choices
(conditional statements)

• Why is this concept useful?
  • Often we want to check if a condition is true and take one action if it is, and another action if the condition is false
  • E.g., If the alternative allele read coverage at a particular location is high enough, annotate the position as a SNP otherwise, annotate the position as reference
# Conditional statement syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
</table>
| If     | if <condition>: # Do something | x = 1  
  if x > 0:  
  print("x is positive") |

**Indentation matters!!!**

Indent the lines of code that belong to the same code block.

Use 4 spaces
Commenting your code

• Why is this concept useful?
  • Makes it easier for—you, your future self, TAs 😊, anyone unfamiliar with your code—to understand what your script is doing

• Comments are human readable text. They are ignored by Python.

• Add comments for
  
<table>
<thead>
<tr>
<th>The how</th>
<th>The why</th>
</tr>
</thead>
<tbody>
<tr>
<td>• What the script does</td>
<td></td>
</tr>
<tr>
<td>• How to run the script</td>
<td></td>
</tr>
<tr>
<td>• What a function does</td>
<td></td>
</tr>
<tr>
<td>• What a block of code does</td>
<td></td>
</tr>
<tr>
<td>• Biological relevance</td>
<td></td>
</tr>
<tr>
<td>• Rationale for design and methods</td>
<td></td>
</tr>
<tr>
<td>• Alternatives</td>
<td></td>
</tr>
</tbody>
</table>

TREAT YOUR CODE LIKE A LAB NOTEBOOK
Commenting rule of thumb

Always code [and comment] as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

-- John Woods
Commenting your code (cont.)

• Commenting is extremely important!

  • Points will be deducted if you do not comment your code
  • If you use code from a resource, e.g., a website, cite it
# Comment syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block comment</strong></td>
<td></td>
</tr>
</tbody>
</table>
| # `<your_comment>` # `<your_comment>` | # Part 5  
# TODO Use overlapping windows to count the  
# dinucleotides in alphabetical order. See the  
# assignment for more information on overlapping  
# windows. |
| **In-line comment** |                                                                         |
| `<code>` # `<your_comment>` | num_genes = 42 # number of diff. expressed genes |
Python modules

• A module is file containing Python definitions and statements for a particular purpose, e.g.,
  • Generating random numbers
  • Plotting
• Modules must be imported at the beginning of the script
  • This loads the variables and functions from the module into your script, e.g.,
    ```python
    import sys
    import random
    ```
• To access a module’s features, type `<module>..<feature>`, e.g.,
  ```python
  sys.exit()
  ```
Random module

• Contains functions for generating random numbers for various distributions

  • TIP: will be useful for assignment 1

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>random.choice</code></td>
<td>Return a random element from a list</td>
</tr>
<tr>
<td><code>random.randint</code></td>
<td>Return a random integer in a given range</td>
</tr>
<tr>
<td><code>random.random</code></td>
<td>Return a random float in the range [0, 1)</td>
</tr>
<tr>
<td><code>Random.seed</code></td>
<td>Initialize the (pseudo) random number generator</td>
</tr>
</tbody>
</table>

https://docs.python.org/3.4/library/random.html
How to repeat yourself
(for loops)

• Why is this useful?
  • Often, you want to do the same thing over and over again
    • Calculate the length of each chromosome in a genome
    • Look up the gene expression value for every gene
    • Align each RNA-seq read to the genome
  • A for loop takes out the monotony of doing something a bazillion times by executing a block of code over and over for you
    • Remember, programmers are lazy!
• A for loop iterates over a collection of things
  • Elements in a list
  • A range of integers
  • Keys in a dictionary
# For loop syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
</table>
| for <counter> in <collection_of_things>:  
  # Do something | ```python
for i in range(0,10):
    print("Hello!")
``` | Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello!  
Hello! |

- The `<counter>` variable is the value of the current item in the collection of things
- You can ignore it
- You can use its value in the loop
- All code in the for loop’s code block is executed at each iteration
- TIP: If you find yourself repeating something over and over, you can probably convert your code to a for loop!

**Indentation matters!!!**

- Indent the lines of code that belong to the same code block
- Use 4 spaces
Which option would you rather do?

A

B

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```
How to repeat yourself (cont.)

• For loops have a close cousin called **while loops**

• The major difference between the 2
  • For loops repeat a block of code a predetermined number of times (really, a collection of things)
  • While loops repeat a block of code **as long as an expression is true**
    • e.g., while it’s snowing, repeat this block of code
    • While loops can turn into **infinite while loops** ➔ the expression is never false so the loop never exits. Be careful!
  • See [http://learnpythonthehardway.org/book/ex33.html](http://learnpythonthehardway.org/book/ex33.html) for a tutorial on while loops
Command-line arguments

• Why are they useful?
  • Passing command-line arguments to a Python script allows a script to be customized

• Example
  • make_nuc.py can create a random sequence of any length
  • If the length wasn’t a command-line argument, the length would be hard-coded
    • To make a 10bp sequence, we would have to 1) edit the script, 2) save the script, and 3) run the script.
    • To make a 100bp sequence, we’d have to 1) edit the script, 2) save the script, and 3) run the script.
  • This is tedious & error-prone
  • Remember: be a lazy programmer!
Can you pass the salt?

I said—

I know! I'm developing a system to pass you arbitrary condiments.

It's been 20 minutes!

It'll save time in the long run!
Command-line arguments

- Python stores the command-line arguments as a list called `sys.argv`
  - `sys.argv[0]` # script name
  - `sys.argv[1]` # 1st command-line argument
  - ...

- **IMPORTANT**: arguments are passed as strings!
  - If the argument is not a string, convert it, e.g., `int()`, `float()`

- `sys.argv` is a list of variables
  - The values of the variables, e.g., the A frequency, are not “plugged in” until the script is run
  - Use the `A_freq` to stand for the A frequency that was passed as a command-line argument
Reading (and writing) to files in Python

Why is this concept useful?

• Often your data is much larger than just a few numbers:
  • Billions of base pairs
  • Millions of sequencing reads
  • Thousands of genes

• It’s may not feasible to write all of this data in your Python script
  • Memory
  • Maintenance

How do we solve this problem?
The solution:

• Store the data in a separate file

• Then, in your Python script
  • **Read** in the data (line by line)
  • Analyze the data
  • **Write** the results to a new output file or print them to the terminal

• When the results are written to a file, other scripts can read in the results file to do more analysis
## Reading a file syntax

### Syntax

```python
with open(<file>) as <file_handle>:
    for <current_line> in open(<file>), 'r'):
        <current_line> = <current_line>.rstrip()
        # Do something
```

### Example

```python
with open(fasta) as f:
    for line in f:
        line = line.rstrip()
        print(line)
```

### Output

`>chr1
ACGTTGAT
ACGTA`
The anatomy of a (simple) script

- The first line should always be 
  `#!/usr/bin/env python3`
- This special line is called a shebang
- The shebang tells the computer how to run the script
- It is NOT a comment
The anatomy of a (simple) script

- This is a special type of comment called a **doc string**, or documentation string.
- Doc strings are used to explain 1) what script does and 2) how to run it.
- ALWAYS include a doc string.
- Doc strings are enclosed in triple quotes, """".

```python
#!/usr/bin/env python3

"""
hello_world.py prints a greeting

Usage: python3 hello_world.py <name>

<name> = Name of person you want to say hello to

"""

# Import modules
import sys

name = sys.argv[1]
print("Hello ", name, "!", sep="")
```
The anatomy of a (simple) script

• This is a comment
• Comments help the reader better understand the code
• Always comment your code!
The anatomy of a (simple) script

• This is an import statement
• An import statement loads variables and functions from an external Python module
• The sys module contains system-specific parameters and functions

```
#!/usr/bin/env python3

"""
hello_world.py prints a greeting
Usage: python3 hello_world.py <name>

<name> = Name of person you want to say hello to

"""

# Import modules
import sys

name = sys.argv[1]
print("Hello ", name, ",!", sep="\"")
```
The anatomy of a (simple) script

- This grabs the command line argument using `sys.argv` and stores it in a variable called `name`
The anatomy of a (simple) script

- This prints a statement to the terminal using the print function
- The first list of arguments are the items to print
- The argument sep="" says do not print a delimiter (i.e., a separator) between the items
- The default separator is a space.
Assignment 1
How to complete & “turn in” assignments

1. Create a separate directory for each assignment

2. Create “submission” and “work” subdirectories
   - Work = scratch work
   - Submission = final version
   - The TAs will only grade content that is in your submission directory

3. Copy the starter scripts and README to your work directory

4. Copy the final version of the files to your submission directory
   - Don’t touch the submission folder again! Timestamps of the files are used to determine if the assignment was turned in on time
README files

- A README.txt file contains information on how to run your code and answers to any of the questions in the assignment
- A template will be provided for each assignment
- Copy the template to your work folder
- Replace the text in {} with your answers
- Leave all other lines alone 😊

---

**A README.txt template**

```plaintext
Question 1:
{nuc_count.py nucleotide count output}
- Comments:
{Things that went wrong or you can not figure out}
- 
```

**A filled out README.txt**

```plaintext
Question 1:
A: 10
C: 15
G: 20
T: 12
- Comments:
The wording for part 2 was confusing.
- 
```
Usage statements in README.txt

• Purpose
  • Tells a user (you, TA, anyone unfamiliar with your) how to run the script
  • Documents how you created your results

• Good practices
  • Write out exactly how you ran the script:
    
    ```
    python3 foo.py 10 bar
    ```
  • AND/OR, write out how to run the script in general, i.e., with placeholders for command-line arguments
    
    ```
    python3 foo.py <#_of_genes> <gene_of_interest>
    ```

• TIP: copy and paste your commands into your README

• TIP: use the command history to view previous commands
Reran analysis pipeline
2 years later
Same result
Assignment 1 TODOs

• Download chr20 via FTP

• You will be given a starter script (nuc_count.py) that counts the total number of A, C, G, T nucleotides
  • Modify the script to calculate the nucleotide frequencies
  • Modify the script to calculate the dinucleotide frequencies

• Modify a starter script (make_seq.py) to generate a random sequence given nucleotide frequencies

• Use make_seq.py to generate random sequence with the same nucleotide frequencies as chr20

• Compare the chr20 di/nucleotide frequencies (observed) with the random model (expected)
Fasta file format

• A standard text-based file format used to define sequences, e.g., nucleotide or peptide sequences
• .fa or .fasta extension
• Each sequence is defined by multiple lines
  • Line 1: Description of sequence. Starts with “>”
  • Lines 2-N: Sequence
• A fasta can contain ≥ 1 sequence

Example fasta file

```
>chr22
ACGGTACGTACCGTAGATNAGTAN

>chr23
ACCGATGTGTGTAGGTCGATNACGTAGTGATGTAT
```
Requirements

• Due next **Friday** (1/26) at 10am

• Your submission folder should contain:
  □ A Python script to count nucleotides (**nuc_count.py**)
  □ A Python script to make a random sequence file (**make_seq.py**)
  □ An output file with a random sequence (**random_seq_1M.txt**)
  □ A README.txt file with instructions on how to run your programs and answers to the questions.

• Remember to comment your script!
CONGRATULATIONS

ON BEING FREAKING AWESOME
<table>
<thead>
<tr>
<th>Topic</th>
<th>Python.org link</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td><a href="https://docs.python.org/3/library/index.html">https://docs.python.org/3/library/index.html</a></td>
</tr>
<tr>
<td>Functions</td>
<td><a href="https://docs.python.org/3/library/functions.html">https://docs.python.org/3/library/functions.html</a></td>
</tr>
<tr>
<td>Integers &amp; floating points numbers</td>
<td><a href="https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex">https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex</a></td>
</tr>
<tr>
<td>Strings</td>
<td><a href="https://docs.python.org/3/library/stdtypes.html#str">https://docs.python.org/3/library/stdtypes.html#str</a></td>
</tr>
<tr>
<td>Lists</td>
<td><a href="https://docs.python.org/3/library/stdtypes.html#list">https://docs.python.org/3/library/stdtypes.html#list</a></td>
</tr>
<tr>
<td>Dictionaries</td>
<td><a href="https://docs.python.org/3/library/stdtypes.html#mapping-types-dict">https://docs.python.org/3/library/stdtypes.html#mapping-types-dict</a></td>
</tr>
</tbody>
</table>
If you’re not completely scared off, come back by 11:20.
Dictionaries: what are they?

• Why are dictionaries useful?
  • We often want to map/associate something with something else, e.g.,
    • A gene and it’s gene expression level

• Dictionaries hold a collection of **key value pairs**
  • The collection is **unordered**
  • A key-value pair is called an **item**

• Keys
  • Can be a strings or numbers
  • **Must be unique**

• Values
  • Can be anything: strings, numbers, even dictionaries!
  • Do not have to be unique

• Also called hash tables & associative arrays
Add syntax and example

- Dictionaries hold a collection of **key value pairs**
  - Keys must be unique and can be anything except a list
  - Values can be anything
- A pair of braces {} creates an empty dictionary.
  - >>> my_dict={}
  - >>> my_dict["first"]="Renee"
  - >>> my_dict["last"]="Sears"
  - >>> my_dict.keys()
  - dict_keys(['last', 'first'])
  - >>> print(my_dict)
  - {'last': 'Sears', 'first': 'Renee'}
  - >>> type(my_dict)
  - <class 'dict'>
Dictionaries: what can I do with them?

<table>
<thead>
<tr>
<th>Task</th>
<th>RNA-seq example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an empty dictionary</td>
<td>Start a brand new analysis</td>
</tr>
</tbody>
</table>
Dictionaries: where can I learn more?

• Python.org tutorial: https://docs.python.org/3.4/tutorial/datastructures.html#dictionaries

• Python.org documentation: https://docs.python.org/3.4/library/stdtypes.html#mapping-types-dict
Python reference for assignment 2: Dictionaries

Create and empty dictionary

```python
<dict> = {}
genes = {}
```

Add a key-value pair to a dictionary

```python
<dict>[<key>] = <value>
genes['BRCA1'] = 'AGTCGT'
```

Look up the value of a key

```python
<value> = <dict>[<key>]
seq = genes['BRCA1']
```

Check if a key is in a dictionary

Returns True or False

```python
<key> in <dict>.keys()
'BRC1A' in genes.keys()
```

Iterate through a dictionary

```python
for <key>, <value> in <dict>.items():
    # Do something
```
Python reference for assignment 2: Strings

Reverse a string
Use extending slicing to take the entire string and step through the string from the end to the beginning

```
<string_reverse> = <string>[:::-1]
```

Extract a substring
End coordinate is exclusive, not inclusive!

```
<substr> = <string>[<start>:<end>]
```

Check if a string starts with a specific prefix
Returns True or False

```
<string>.startswith(<prefix>)
```

Translate a string

```
<translation_map> = str.maketrans(<dictionary of letters to translate>)
<string_translated> = <string>.translate(<translation_map>)
```

```
lowercase_map = str.maketrans({'X': 'x', 'Y': 'y'})
lower_seq = seq.translate(lowercase_map)
```