• Print out slides
• bb guest session
• Open up sublime text
• Cyberduck
Lab 1: Introduction to Python Programming
A few preliminary words…
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
Overview

- Logistics
- Getting Started
- Intro to Unix
- Intro to Python
- Assignment 1
Logistics

• **Office Hours**: Wednesdays after class (11:30-12:30 pm) in the 4th floor classroom 4515 McKinley
• **Contact TAs**: bio5488wustl@gmail.com
• Register for 4 credits
• Course website: [http://genetics.wustl.edu/bio5488/](http://genetics.wustl.edu/bio5488/)
• Bring your laptop to every lab
• **NO extensions** on homeworks
• **Late penalty** is -50% per day
Assignments

• Assignments are posted on the course website Wednesdays at 10am
• Assignments are due the following Friday at 10am (before lab)

• 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 😊
<table>
<thead>
<tr>
<th>Wed</th>
<th>Thurs</th>
<th>Fri</th>
<th>Sat</th>
<th>Sun</th>
<th>Mon</th>
<th>Tue</th>
<th>Wed</th>
</tr>
</thead>
</table>

- **HW released**
- **Class discussion & work time** 10-11:30am
- **Office hours** 11:30-12:30pm
- **Tutoring session** 3:15-5pm
- **HW due** 10am
## Schedule (cont.)

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Released</th>
<th>Due</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1/20</td>
<td>1/29</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>1/27</td>
<td>2/3</td>
<td>Sequence Comparison</td>
</tr>
<tr>
<td>3</td>
<td>2/3</td>
<td>2/10</td>
<td>Next Gen Sequencing</td>
</tr>
<tr>
<td>4</td>
<td>2/10</td>
<td>2/17</td>
<td>Gene Expression</td>
</tr>
<tr>
<td>5</td>
<td>2/17</td>
<td>2/24</td>
<td>Epigenomics</td>
</tr>
<tr>
<td>6</td>
<td>2/24</td>
<td>3/2</td>
<td>Synthetic Gene Assembly</td>
</tr>
<tr>
<td>7</td>
<td>3/2</td>
<td>3/23</td>
<td>Motif Finding</td>
</tr>
<tr>
<td>8</td>
<td>3/2</td>
<td>3/23</td>
<td>Metagenomics</td>
</tr>
<tr>
<td>9</td>
<td>3/23</td>
<td>3/30</td>
<td>Genetic Variation</td>
</tr>
<tr>
<td>10</td>
<td>3/30</td>
<td>4/6</td>
<td>Wright-Fisher Model</td>
</tr>
<tr>
<td>11</td>
<td>4/6</td>
<td>4/13</td>
<td>Substitution Rates</td>
</tr>
<tr>
<td>12</td>
<td>4/13</td>
<td>4/20</td>
<td>TBD</td>
</tr>
<tr>
<td>13</td>
<td>4/20</td>
<td>4/27</td>
<td>Cis Regulatory Evolution</td>
</tr>
</tbody>
</table>

Deadline extended: due next Friday

2 labs over spring break
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
  • 50% reduction per day

• 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
Remote computers

• We will be doing all of our work on a remote computer with the hostname 45.62.227.83

• This is a Unix-based computer that we can securely connect to through a protocol called secure shell (SSH).
What is the shell?

- The **shell** is a program that takes commands from the keyboard and gives them to the operating system to execute
  - There are many different shell programs
  - We’ll be using the most common shell: the **Bourne-Again SHell (bash)**
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 or Ubuntu (Windows) and Terminal (Mac)
How to log onto the remote computer

Live Demo
Why should I learn how to use shells and terminals?

- CLIs are common in scientific computing → get used to them!
- The shell is a really powerful way of interacting with your computer → become a super user!
How to log onto the remote computer (Putty users)

1. Launch Putty
2. In the host name field, enter username@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/Ubuntu users)

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

2. SSH to the remote computer. Type:

```
ssh <username>@genomic.wustl.edu
```

where `<username>` is replaced with your username

3. A security message may be printed. Type `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
Cyberduck

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
The file system

• The **file system** is the part of the operating system (OS) responsible for managing files and folders
  • In Unix, folders are called **directories**.

• Unix keeps files arranged in a hierarchical structure
  • The topmost directory is called the **root directory**
  • Each directory can contain
    • Files
    • Subdirectories

• You will always be “in” a directory
  • When you open a terminal you will be in your own **home directory**.
  • Only you can modify things in your home directory
Determining where you are

(\textit{pwd})

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

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

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

(\texttt{ls})

• To list the contents of a directory:
  \begin{verbatim}
  $ \texttt{ls}
  assignment1 foo
  \end{verbatim}

• \texttt{ls} stands for \texttt{list directory contents}
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>
```

output

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 make directory

• 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

\textit{(mv)}

• To rename/move a file/directory

\begin{verbatim}
$ mv -i <original_filename> <new_filename>
\end{verbatim}

where

\begin{align*}
<\textit{original\_filename}> &= \text{name of file/dir you want to rename} \\
<\textit{new\_filename}> &= \text{name you want to rename it to}
\end{align*}

• \textit{mv} stands for \textit{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

\( (\text{rm}) \)

• To delete a file
  
  $ \text{rm} \ <\text{file_to_delete}>$
  
  where
  
  \(<\text{file_to_delete}> = \text{name of the file you want to delete}\)

• To delete a directory
  
  $ \text{rm} \ -r \ -i \ <\text{directory_to_delete}>$
  
  where
  
  \(<\text{directory_to_delete}> = \text{name of the directory you want to delete}\)

• \text{rm} \text{ stands for} \text{r} \text{e} \text{m} \text{ove files/directories}

\[\text{TIP: Check that you’re going to delete the correct files by first testing with ‘ls’ and then committing to ‘rm’}\]

\[\text{IMPORTANT: there is no recycle bin/trash folder on Unix!!}\]

\[\text{Once you delete something, it is gone forever.}\]
\[\text{Be very careful when you use rm!!}\]
Exercise: deleting things

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

• *Save* the output to a file
  
  ```bash
  $ <cmd> > <output_file>
  ```
  
  where
  
  - `<cmd>` = command
  - `<output_file>` = name of output file

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

• *Append* the output to the end of a file
  
  ```bash
  $ <cmd> >> <output_file>
  ```

There are 2 “>”
Learning more about a command (man)

• To view a command’s documentation
  
  $ man <cmd>

  where

  $<cmd> = command

• man stands for manual page

• Use the ↑ and ↓ 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
  
  $\text{ctrl} + C$

• Temporarily stop a command
  
  $\text{ctrl} + Z$
Unix commands cheatsheet--your new bestie

### File Commands
- `ls`: directory listing
- `ls -al`: formatted listing with hidden files
- `cd`: change directory
- `pwd`: show current directory
- `mkdir`: create a directory
- `rm`: delete file
- `rm -r`: delete directory
- `cp`: copy file
- `mv`: move or rename file
- `ln`: create symbolic link
- `cat`: display file
- `more`: display file
- `tail`: display the last 10 lines of file
- `head`: display the first 10 lines of file

### File Permissions
- `chmod`: change permissions
  - `chmod octal file`: change permissions
  - `chmod 777 file`:
    - 4 - read (r)
    - 2 - write (w)
    - 1 - execute (x)
- Examples:
  - `chmod 777 file` - read, write, execute for all
  - `chmod 755 file` - rw for owner, rx for group and world

### Process Management
- `ps`: display your currently active processes
- `top`: display all running processes
- `kill`: kill process id
- `killall`: kill all processes named
- `bg`: lists stopped or background jobs; resume a stopped job in the background
- `fg`: brings the most recent job to foreground
- `fg` `-n`: brings job `n` to the foreground

### SSH
- `ssh`: connect to host as user
  - `ssh user@host`:
- `ssh -p port user@host`: connect to host on port `port` as user
- `ssh-copy-id user@host`: add your key to host for user to enable a keyed or passwordless login

### Shortcuts
- `Ctrl+C`: halts the current command
- `Ctrl+Z`: stops the current command, resume with `fg`
- `Ctrl+D`: log out of current session, similar to `exit`
- `Ctrl+W`: erases one word in the current line
- `Ctrl+U`: erases the whole line
- `Ctrl+R`: type to bring up a recent command
- `!!`: repeats the last command
- `exit`: log out of current session

### Searching
- `grep pattern files`: search for `pattern` in `files`
- `grep -r pattern dir`: search recursively for `pattern` in `dir`
- `command | grep pattern`: search for `pattern` in the output of `command`
- `locate file`: find all instances of `file`
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:
  - Interactive mode
    - Start Interactive mode via `python3`
  - Normal mode
    - Execute a script via `python3 <script name>`

- Live Demo
Interacting with Python

There are 2 main ways of interacting with Python:

<table>
<thead>
<tr>
<th>Interactive mode</th>
<th>Description</th>
<th>Benefits</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Takes single user inputs, evaluates them, and returns the result to the user (read–eval–print loop (REPL))</td>
<td>• Use as a sandbox: explore new features • Easy to write quick “throw away” scripts • Useful for debugging • Use it as a calculator! • Run long complicated programs</td>
<td>This is Python’s command prompt. It means, “I’m ready for a command!” Don’t type the “&gt;&gt;&gt;”</td>
</tr>
</tbody>
</table>
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><strong>Integers</strong></td>
<td>A whole number</td>
<td>x = 10</td>
</tr>
<tr>
<td><strong>Floats</strong></td>
<td>A real number</td>
<td>x = 5.6</td>
</tr>
<tr>
<td><strong>Strings</strong></td>
<td>Text (1 or more <strong>characters</strong>)</td>
<td>x = “Genomics”</td>
</tr>
<tr>
<td><strong>Booleans</strong></td>
<td>A binary outcome: true or false</td>
<td>x = True</td>
</tr>
</tbody>
</table>
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 $HOXA1$ gene was differentially expressed

• What is the type for each variable?

Cheatsheet

<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 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>

You can use single quotes or double quotes.
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 **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

• Python.org documentation: 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 5</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>&gt;&gt;&gt; 2 - 3 -1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>&gt;&gt;&gt; 2 * 3 6</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>&gt;&gt;&gt; 2 / 3 0.6666666666666666</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; '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>&gt;&gt;&gt;&gt; 2 &lt; 3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>&gt;&gt;&gt;&gt; 2 &lt;= 3</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>&gt;&gt;&gt;&gt; 2 &gt; 3</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>&gt;&gt;&gt;&gt; 2 &gt;= 3</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>&gt;&gt;&gt;&gt; 2 == 3</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>&gt;&gt;&gt;&gt; 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><strong>and</strong></td>
<td>Return True if <em>both</em> arguments are true</td>
<td>&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>Call a function that takes no arguments</td>
<td><code>&lt;function_name&gt;()</code></td>
<td><code>sys.exit()</code></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()`                                                    |
| `<str>.lower()` | • Returns the string with all letters lowercased                 |                                                                         |
| `<str>.find(<pattern>)` | • Returns the first index of <pattern> in the string               | `>>> x.find('nom')`                                                   |
| `<str>.count(<pattern>)` | • Returns the number of times <pattern> is found                  | `>>> x.count('g')`                                                   |
| `<str>[<index>]` | • Returns the letter at the <index>th position                  |                                                                         |

[https://docs.python.org/3.4/library/stdtypes.html#string-methods](https://docs.python.org/3.4/library/stdtypes.html#string-methods)
[https://docs.python.org/3/library/stdtypes.html#str](https://docs.python.org/3/library/stdtypes.html#str)
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  

```python
x = 1
if x > 0:
    if x > 0:
        print("x is positive")
```
|        |         | 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

  The how
  • What the script does
  • How to run the script
  • What a function does
  • What a block of code does

  The why
  • Biological relevance
  • Rationale for design and methods
  • Alternatives

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>
</table>
| **Block comment**    | # <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.,

    import sys
    import random

• To access a module’s features, type `<module>..<feature>`, e.g.,
  `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>
<tbody>
<tr>
<td>for &lt;counter&gt; in &lt;collection_of_things&gt;:</td>
<td>for i in range(0,10): print(&quot;Hello!&quot;)</td>
<td>Hello! Hello! Hello! Hello! Hello! Hello! Hello! Hello! Hello! Hello!</td>
</tr>
</tbody>
</table>

- 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?
Reading (and writing) to files in Python

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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\begin{verbatim} with open(&lt;file&gt;) as &lt;file_handle&gt;: for &lt;current_line&gt; in open(&lt;file&gt;) , 'r'): &lt;current_line&gt; = &lt;current_line&gt;.rstrip() # Do something \end{verbatim}</td>
<td>\begin{verbatim} with open(fasta) as f: for line in f: line = line.rstrip() print(line) \end{verbatim}</td>
</tr>
</tbody>
</table>

### 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
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**

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

**A filled out README.txt**

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
ACCGATGTGTGTAGGTACGTNACG
TAGTGATGTAT
```
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
# Python documentation

<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 its 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
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()
'BRCAL1' 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)
```