Lab 1: Introduction to Python Programming

1/20/17

Slide credits:

Nicole Rockweiler!

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, discussion groups, 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 (11:30am-12:30 pm) in the 4th floor classroom 4515 McKinley/area outside the classroom and by appointment
- Come to tutoring sessions
  - Tuesdays 5:30-7pm in 6001B* Scott McKinley Building
  - *4/4 will be in 5001B
  - FREE FOOD!!
- Use the google docs to ask/answer questions - https://docs.google.com/spreadsheets/d/11KW_lu9mE59LBtF0X8EtrCJfHQZ22fQwz8AC3AMZSs8/edit?usp=sharing
- Email bio5488wustl@gmail.com
- Work in groups

Where to get help
(a.k.a. how to maintain your sanity)

Assignments

- Assignments are posted on the course website Wednesdays at 10am
- Assignments are due the following Wednesday 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 ☺

Schedule

Wed	Thurs	Fri	Sat	Sun	Mon	Tue	Wed
HW released		Class discussion & work time 10-11:30am			Office hours 11:30-12:30pm	Tutoring session 5-7:30pm	HW due 10am

Schedule (cont.)

Assignment	Released	Due	Topic
1	1/18	1/27	Introduction
2	1/25	2/1	Sequence Comparison
3	2/1	2/8	Next Gen Sequencing
4	2/8	2/15	Gene Expression
5	2/15	2/22	Epigenomics
6	2/22	3/1	Motif Finding
7	3/1	3/22	Synthetic Gene Assembly
8	3/1	3/22	Metagenomics
9	3/22	3/29	Genetic Variation
10	3/29	4/5	Wright-Fisher Model
11	4/5	4/12	TBD
12	4/12	4/19	Substitution Rates
13	4/19	4/26	Cis Regulatory Evolution

2 labs over spring break

Assignment policies

- See the Course Information → Assignment policies document on 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 (early) to request an extension
- Auditors
  - We'll give comments on your programs, but won't grade the short answer questions
  - Same late policy applies
- Collaboration
  - Group work is encouraged, but plagiarism is unacceptable
  - Try to "Google it" first
  - Cite your sources
- Work on the assignment before coming to lab

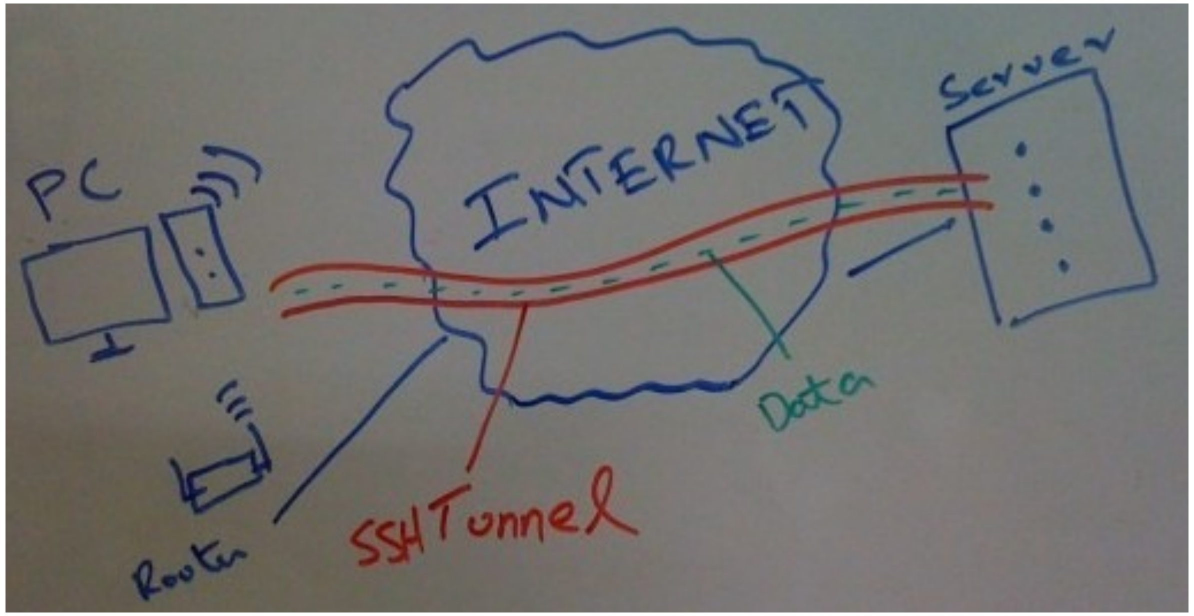
Grading

- 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** `genomic.wustl.edu`
- This is a Unix-based computer that we can securely connect to through a protocol called **secure shell** (**SSH**).

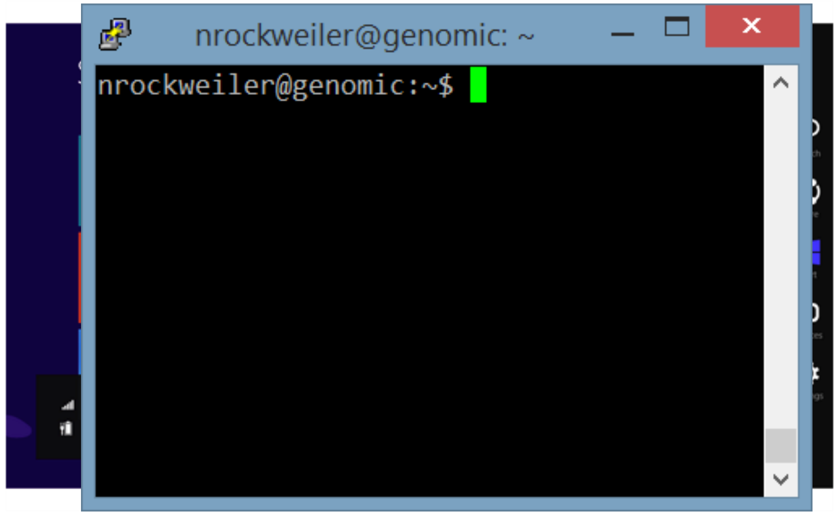


What is the shell?

- 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**)

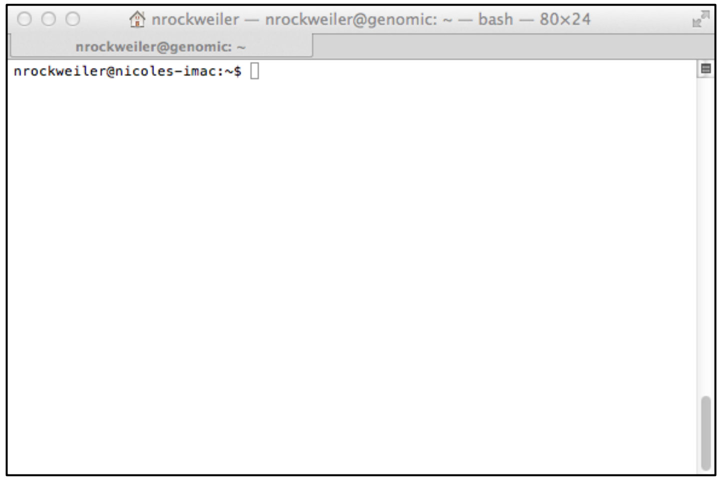
How do I access the shell?

- Most of us are familiar with **graphical user interfaces** (**GUI**) to control our computers
- Another way is with **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)



```
nrockweiler@genomic:~$
```

A Windows GUI
A PuTTY window



```
nrockweiler@nicoles-imac:~$
```

A Terminal window

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!

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“

This is called the **command prompt**. It means, “I’m ready for a command!” Don’t type the “$.”

Don’t type the “<>”

Template:

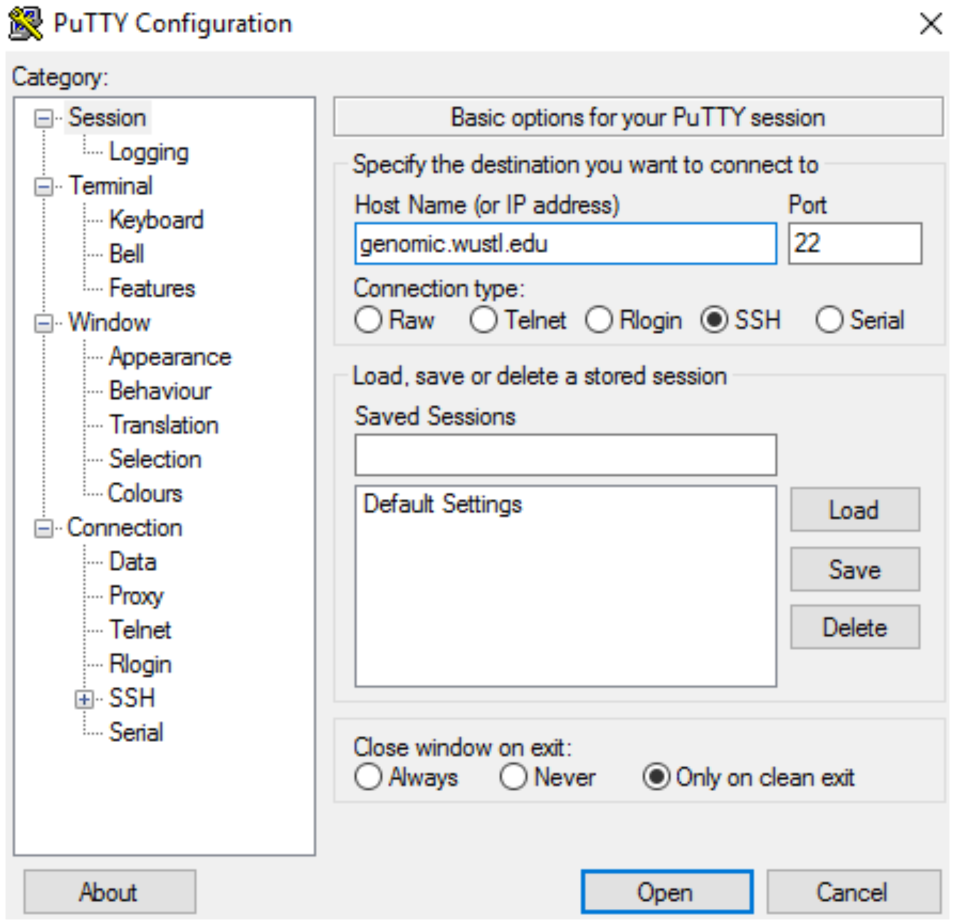
```
$ type_me_exactly <modify_me>
output
```

Example:

```
$ ls <assignment>
README.txt
```

How to log onto the remote computer (Windows users)

1. Launch Putty
2. In the host name field, enter `genomic.wustl.edu`
3. Enter a session nickname, e.g., bio5488
4. Click Save
5. Click Open


PuTTY Configuration

Category:

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): genomic.wustl.edu
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit: Always Never [✓ Only on clean exit]

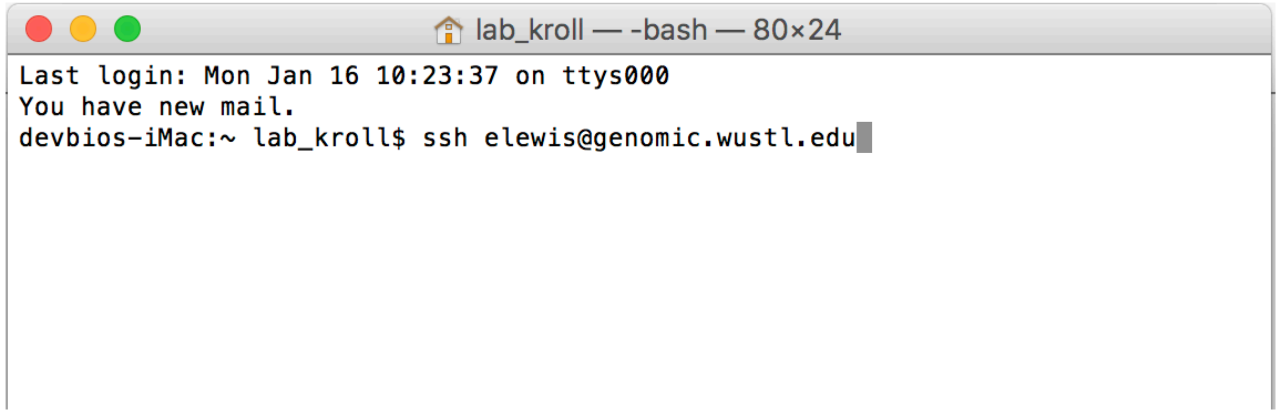
How to log onto the remote computer (Mac users)

1. Open Terminal (found in `/Applications/Utilities`)

2. SSH to the remote computer. Type:

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

where `<username>` is replaced with your username

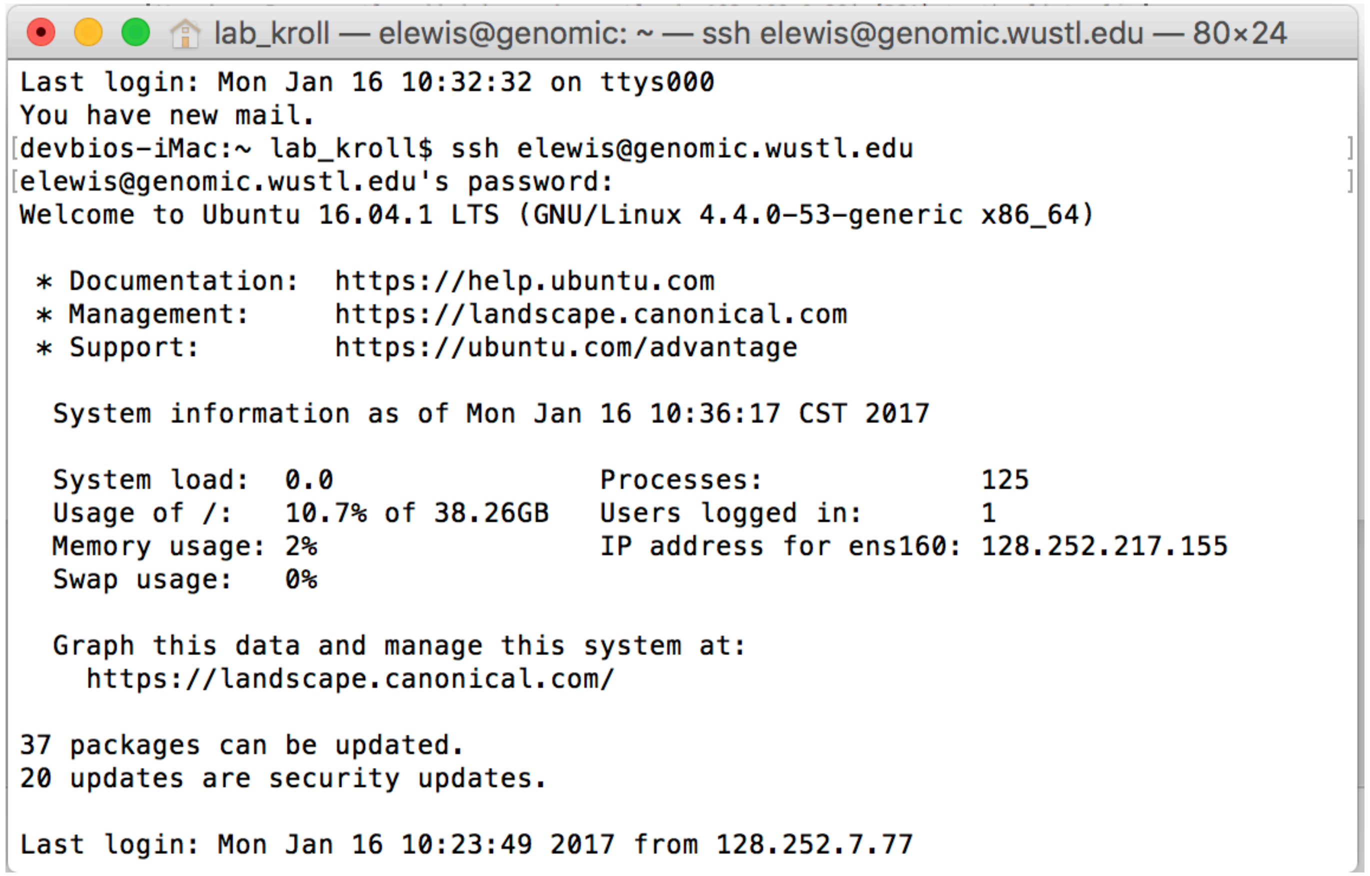


```
Last login: Mon Jan 16 10:23:37 on ttys000
You have new mail.
devbios-iMac:~ lab_kroll$ ssh elewis@genomic.wustl.edu
```

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.


lab_kroll — elewis@genomic: ~ — ssh elewis@genomic.wustl.edu — 80×24

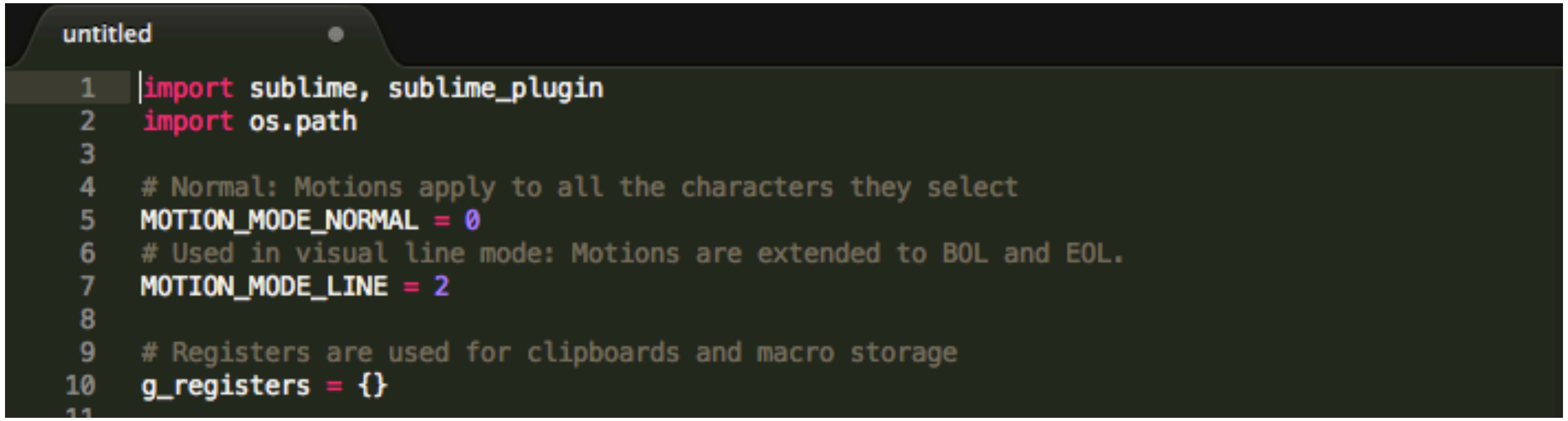
```
Last login: Mon Jan 16 10:32:32 on ttys000
You have new mail.
[devbios-iMac:~ lab_kroll$ ssh elewis@genomic.wustl.edu
[elewis@genomic.wustl.edu's password:
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-53-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Jan 16 10:36:17 CST 2017

  System load:  0.0                Processes:             125
  Usage of /:   10.7% of 38.26GB   Users logged in:       1
  Memory usage: 2%                 IP address for ens160: 128.252.217.155
  Swap usage:   0%

  Graph this data and manage this system at:
    https://landscape.canonical.com/

37 packages can be updated.
20 updates are security updates.

Last login: Mon Jan 16 10:23:49 2017 from 128.252.7.77
```

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.

Sublime Text

- Sublime Text is a **text editor** for writing and editing scripts
- We'll use Sublime to edit both local and remote files
- Documentation: http://www.sublimetext.com/support



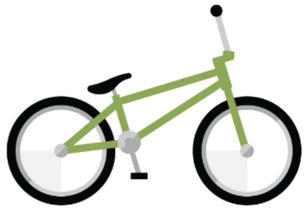
```
import sublime, sublime_plugin
import os.path

# Normal: Motions apply to all the characters they select
MOTION_MODE_NORMAL = 0
# Used in visual line mode: Motions are extended to BOL and EOL.
MOTION_MODE_LINE = 2

# Registers are used for clipboards and macro storage
g_registers = {}
```

Cyberduck

- Cyberduck is a secure **file transfer client** and will allow you to transfer files from your local computer to a remote computer

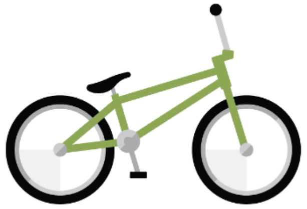
Exercise: setting up Cyberduck

- 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 Cyberduck/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



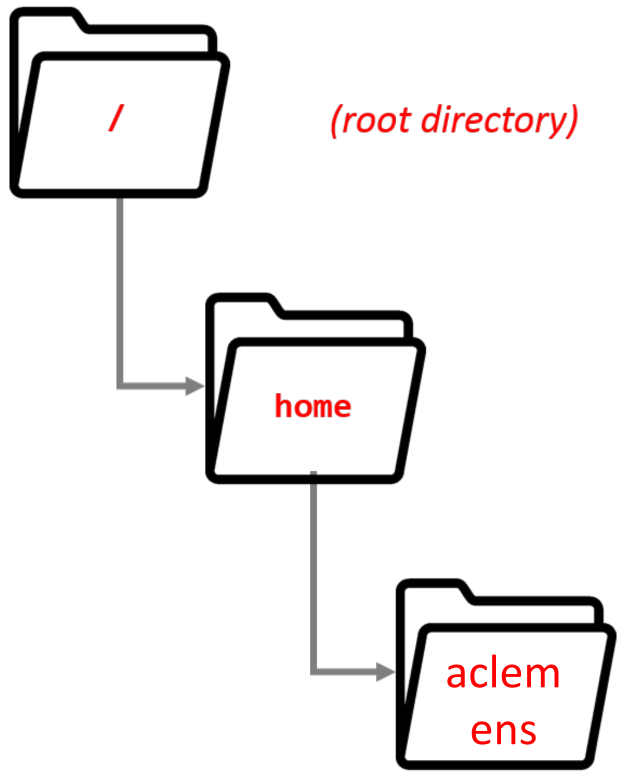
Exercise: editing remote files with Sublime Text and Cyberduck

- 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

Basic Unix

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 (**pwd**)

- If you get lost in the file system, you can determine where you are by typing:

```
$ pwd
/home/aclemens
```

- pwd stands for print working directory
- pwd prints the full **path** of the **current working directory**

Listing directory contents (`ls`)

- To list the contents of a directory:

```
$ ls
assignment1 foo
```

- ls stands for list directory contents

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`)

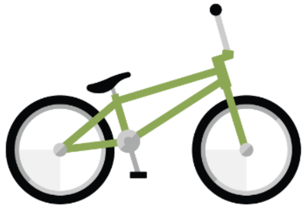
- To make a directory

  ```
  $ mkdir <new_directory_name>
  ```

  where

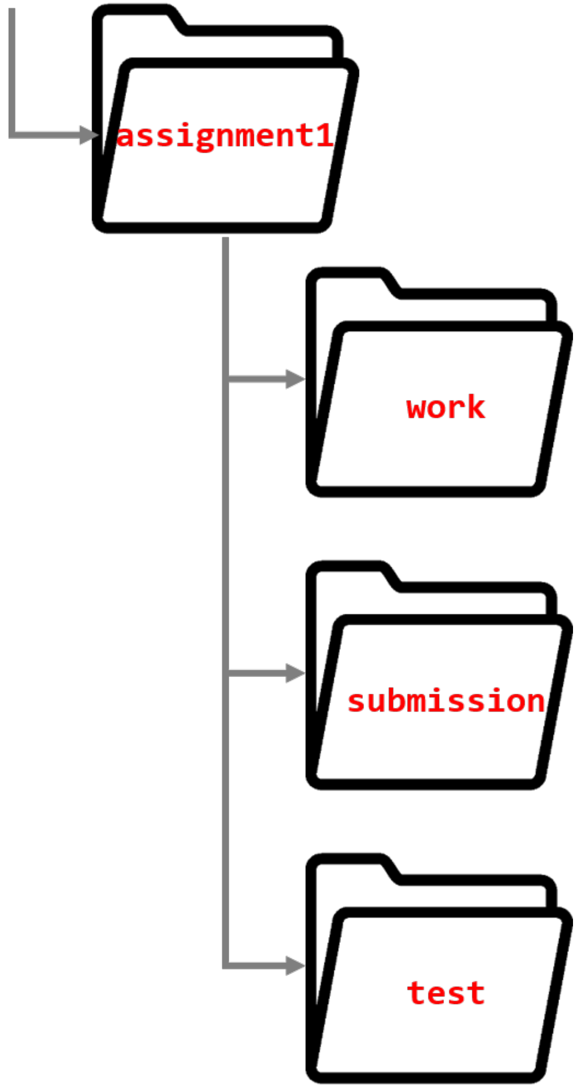
  `<new_directory_name>` = name of the directory to create

- 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 (interactive)

- 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

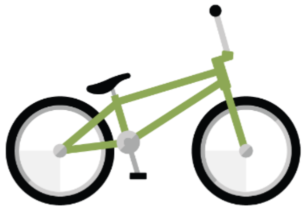
- To create a copy of file *and keep the name the same*

```
$ 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 (rm)

- To delete a file

  ```
  $ rm <file_to_delete>
  ```

  where

  `<file_to_delete>` = name of the file you want to delete

- To delete a directory

  ```
  $ rm -r -i <directory_to_delete>
  ```

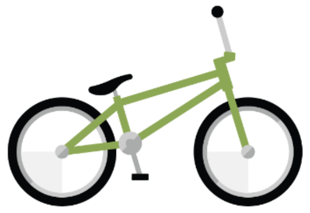
  where

  `<directory_to_delete>` = name of the directory you want to delete

- rm stands for remove files/directories

TIP: Check that you're going to delete the correct files by first testing with 'ls' and then committing to 'rm'

IMPORTANT: there is no recycle bin/trash folder on Unix!!
Once you delete something, it is gone forever.
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

```
$ <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

```
$ <cmd> >> <output_file>
```

  There are 2 “>”

Learning more about a command (`man`)

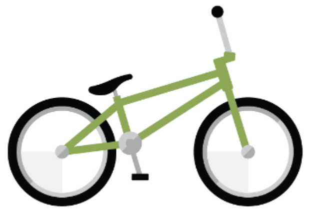
- To view a command's documentation

  ```
  $ 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
```

Getting yourself out of trouble

- Abort a command

  ctrl + C

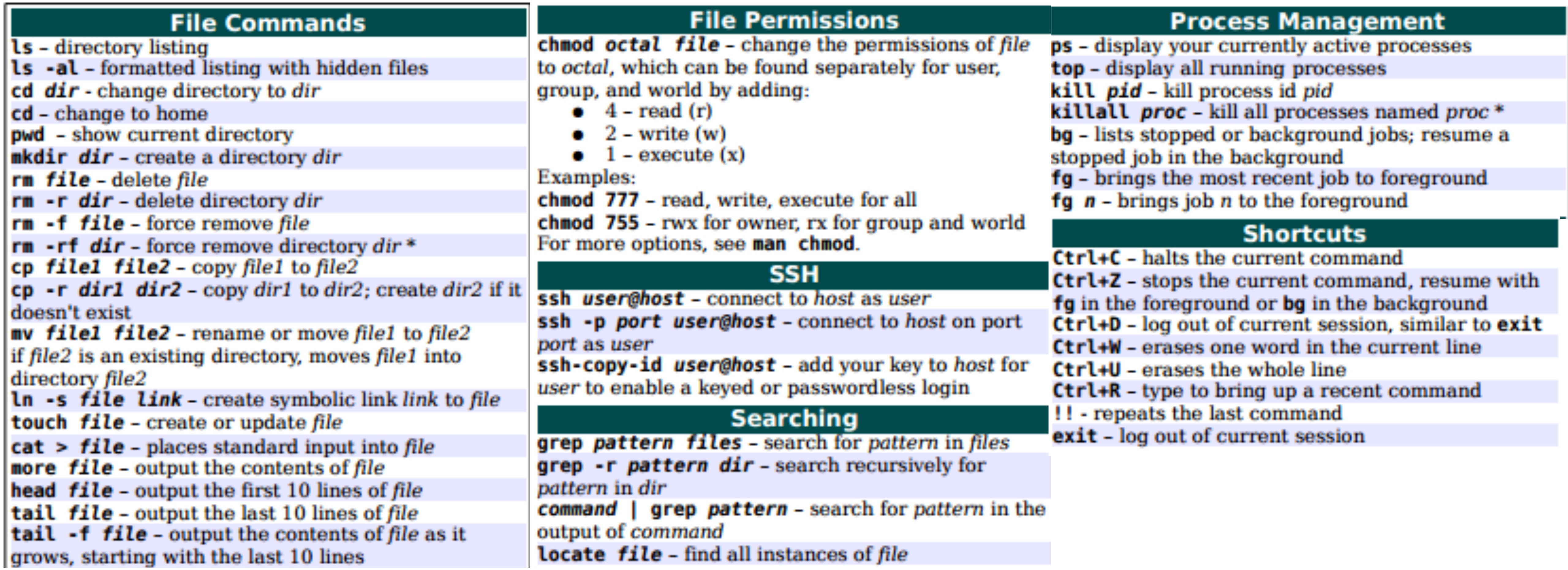
- Temporarily stop a command

  ctrl + Z

- Resume a stopped job

```
$ fg <job_id>
```

Unix commands cheatsheet--your new bestie



File Commands

ls – directory listing
ls -al – formatted listing with hidden files
cd *dir* - change directory to *dir*
cd – change to home
pwd – show current directory
mkdir *dir* – create a directory *dir*
rm *file* – delete *file*
rm -r *dir* – delete directory *dir*
rm -f *file* – force remove *file*
rm -rf *dir* – force remove directory *dir* *
cp *file1 file2* – copy *file1* to *file2*
cp -r *dir1 dir2* – copy *dir1* to *dir2*; create *dir2* if it doesn't exist
mv *file1 file2* – rename or move *file1* to *file2* if *file2* is an existing directory, moves *file1* into directory *file2*
ln -s *file link* – create symbolic link *link* to *file*
touch *file* – create or update *file*
cat > *file* – places standard input into *file*
more *file* – output the contents of *file*
head *file* – output the first 10 lines of *file*
tail *file* – output the last 10 lines of *file*
tail -f *file* – output the contents of *file* as it grows, starting with the last 10 lines

File Permissions

chmod *octal file* – change the permissions of *file* to *octal*, which can be found separately for user, group, and world by adding:

- 4 – read (r)
- 2 – write (w)
- 1 – execute (x)

Examples:
chmod 777 – read, write, execute for all
chmod 755 – rwx for owner, rx for group and world
For more options, see **man chmod**.

SSH

ssh *user@host* – connect to *host* as *user*
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

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*

Process Management

ps – display your currently active processes
top – display all running processes
kill *pid* – kill process id *pid*
killall *proc* – kill all processes named *proc* *
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

Shortcuts

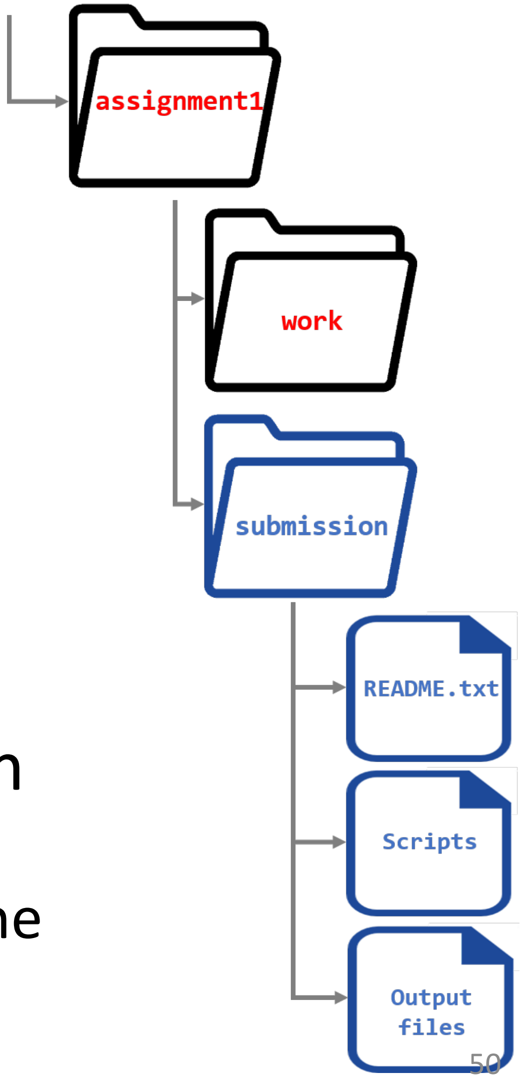
Ctrl+C – halts the current command
Ctrl+Z – stops the current command, resume with **fg** in the foreground or **bg** in the background
Ctrl+D – log out of current session, similar to **exit**
Ctrl+W – erases one word in the current line
Ctrl+U – erases the whole line
Ctrl+R – type to bring up a recent command
!! - repeats the last command
exit – log out of current session

https://ubuntudanmark.dk/filer/fwunixref.pdf

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.xt

```
Question 1:
A: 10
C: 15
G: 20
T: 12
-
Comments:
The wording for part 2 was confusing.
-
```

Usage statements in README.txt

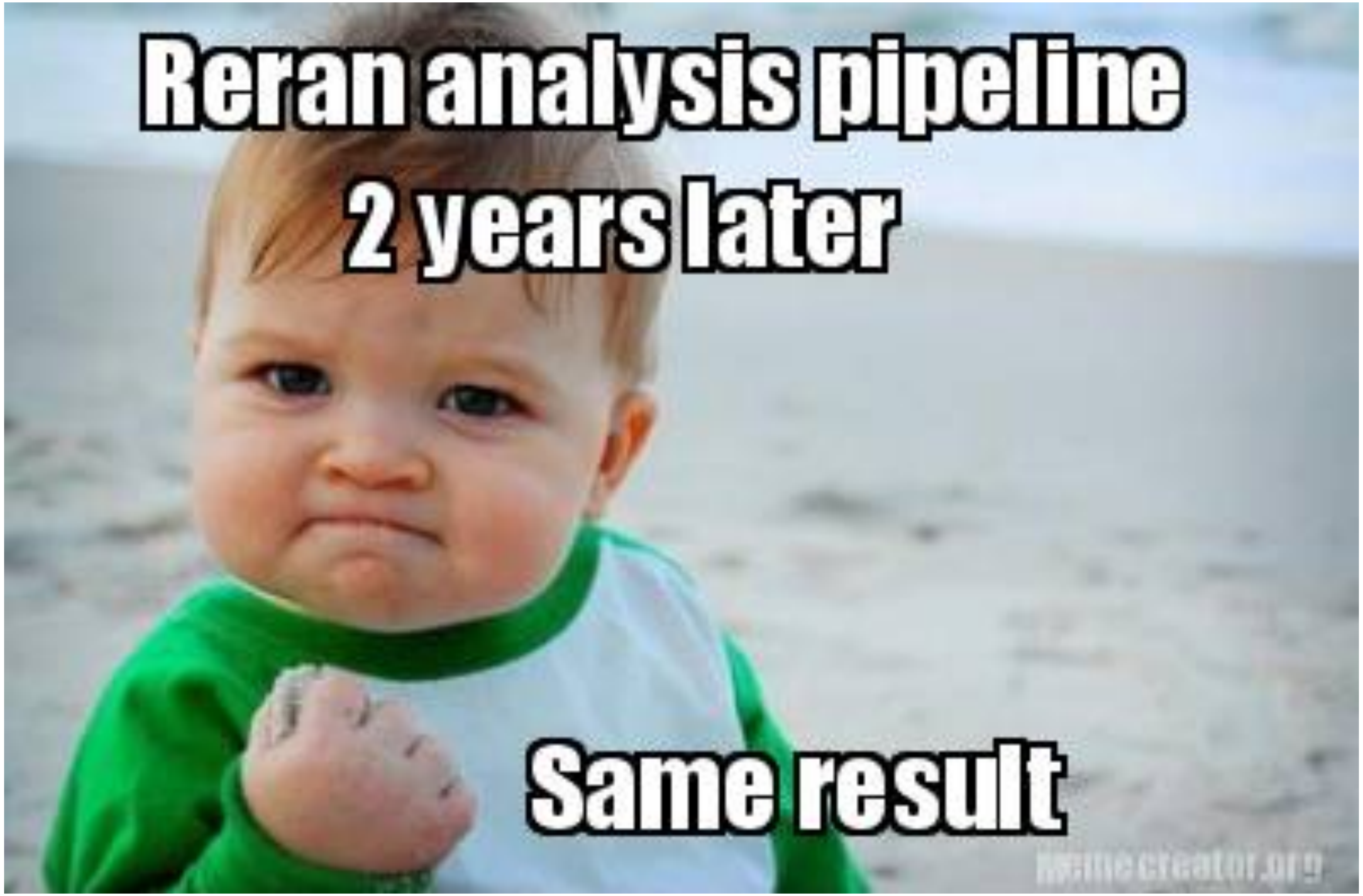
- Purpose
  - Tells a user (you, TA, anyone unfamiliar with your) how to run the script
  - Documents how you created your results
- Good practices
  - Write out exactly how you ran the script:

    ```
    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 (uparrow)


Reran analysis pipeline
2 years later
Same result
memecreator.org

Assignment 1 TODOs

- Download chr20 via **FTP** (here we use wget)
- 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

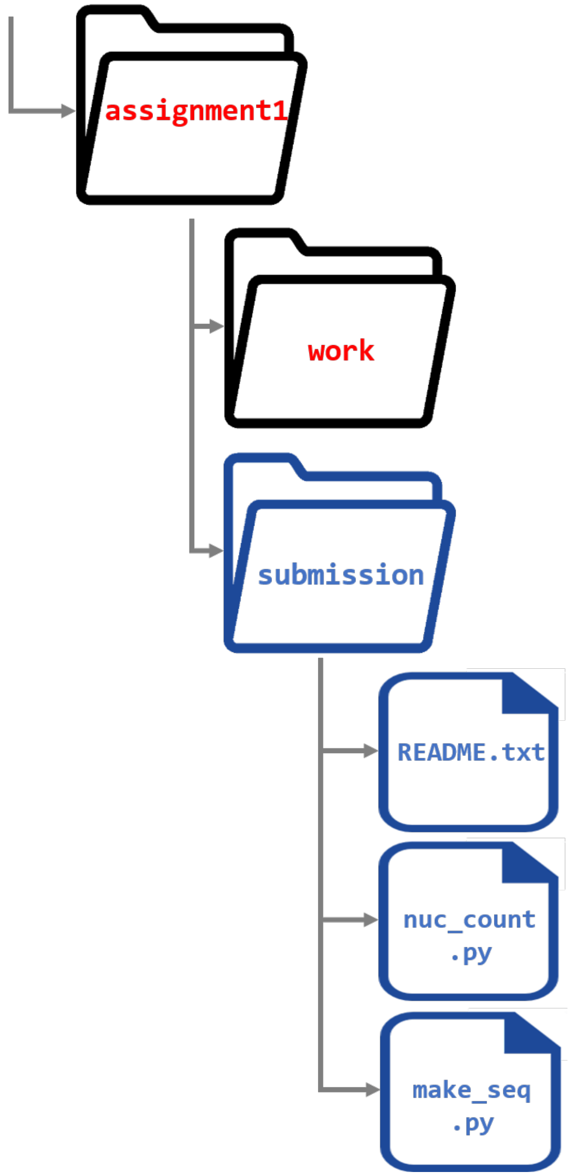
- A standard text-based file format used to define sequences, e.g., nucleotide or peptide sequences
- .fa or .fasta extension
- Each sequence is defined by multiple lines
  - Line 1: Description of sequence. Starts with “>”
  - Lines 2-N: Sequence
- A fasta can contain ≥ 1 sequence

Example fasta file

```
>chr22
ACGGTACGTACCGTAGATNAGTAN
>chr23
ACCGATGTGTGTAGGTACGTNACG
TAGTGATGTAT
```

Requirements

- Due next <u>Friday</u> (1/27) 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!



Python basics

Recycling Nicole's slides from year 2016*

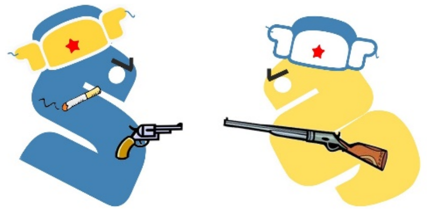
What is Python?

- Python is a widely used programming language
- First implemented 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

Van Rossum is known as a "Benevolent Dictator For Life" (BDFL)

Which Python?

- There are 2 widely used versions of Python: Python2.7 and Python3.x
- We'll use Python**3**
- Many help forums still refer to Python2, so make sure you're aware which version is being referenced



Interacting with Python

There are 2 main ways of interacting with Python:

	Interactive mode	**Normal mode**
Description	Takes single user inputs, evaluates them, and returns the result to the user (**read–eval–print loop** (**REPL**))	Execute a Python **script** on the Unix command prompt
Benefits	• 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 • The script contains all of the commands

Usage	`$ python3`
	`Python 3.4.0 (default, Apr 11 2014, 13:05:11) [GCC 4.8.2] on linux2 Type "help", "copyright", "credits" or "license" for more information.`
	`>>>`
		`$ python3 <script.py>`

This is Python’s command prompt. It means, “I’m ready for a command!” Don’t type the “>>>”

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:

Type	Description	Example
Integers	A whole number	x = 10
Floats	A real number	x = 5.6
Strings	Text (1 or more **characters**)	x = “Genomics”
Booleans	A binary outcome: true or false	x = True

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'>
```

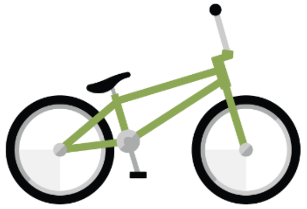
- IMPORTANT: the variable name must be on the <u>left hand side</u> of the =

```
>>> 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

Type	Description	Example
Integers	A whole number	x = 10
Floats	A real number	x = 5.6
Strings	Text (1 or more **characters**)	x = "Genomics"
Booleans	A binary outcome: true or false	x = True

You can use single quotes or double quotes

Collections of things

- Why is this concept useful?
  - We often have collections of things, e.g.,
    - A list of genes in a pathway
    - A list of gene fusions in a cancer cell line
    - A list of probe IDs on a microarray and their intensity value
  - We *could* store each item in a collection in a separate variable, e.g.,

```
gene1 = ‘SUCLA2’
gene2 = ‘SDHD’
...
```

  - A better strategy is to put all of the items in one container
- Python has several types of containers
  - **List** (similar to arrays)
  - **Set**
  - **Dictionary**

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.

Syntax	Example	Output
Create a list		
`<list_name> = [<item1>, <item2>]`	`genes = ['SUCLA2', 'SDHD']`	
Index a list		
`<listname>[<position>]`	`genes[1]`	'SDHD'

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:

Operator	Description	Example
`+`	Addition	`>>> 2 + 3` `5`
`-`	Subtraction	`>>> 2 - 3` `-1`
`*`	Multiplication	`>>> 2 * 3` `6`
`/`	Division	`>>> 2 / 3` `0.6666666666666666`

Operators (cont.)

You can also use operators on strings!

Operator	Description	Example

+	Combine strings together	`>>> 'Bio' + '5488'`
		`'Bio5488'`
		`>>> 'Bio' + 5488`
		`Traceback (most recent call last):`
		`  File "<stdin>", line 1, in <module>`
		`TypeError: Can't convert 'int' object to str implicitly`

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

Operator	Description	Example
`<`	Less than	`>>> 2 < 3` `True`
`<=`	Less than or equal to	`>>> 2 <= 3` `True`
`>`	Greater than	`>>> 2 > 3` `False`
`>=`	Greater than or equal to	`>>> 2 >= 3` `False`
`==`	Equal to	`>>> 2 == 3` `False`
`!=`	Not equal to	`>>> 2 != 3` `True`

== 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

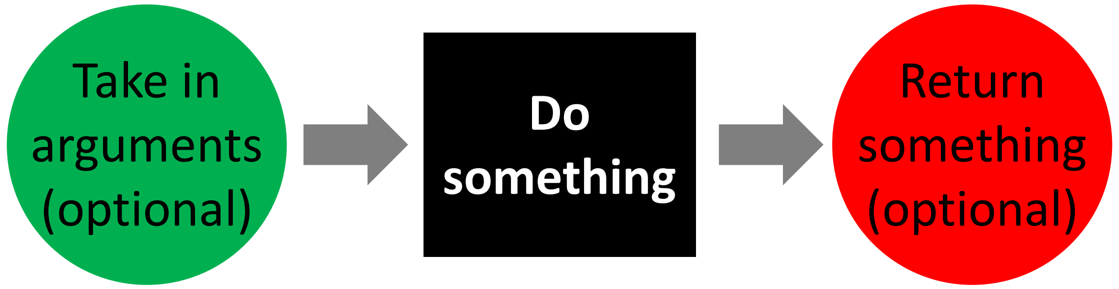
Operator	Description	Example

and	Return True if *both* arguments are true	`>>> True and True`
		`True`
		`>>> True and False`
		`False`

or	Return True if *either* arguments are true	`>>> True or False`
		`True`
		`>>> False or False`
		`False`

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?

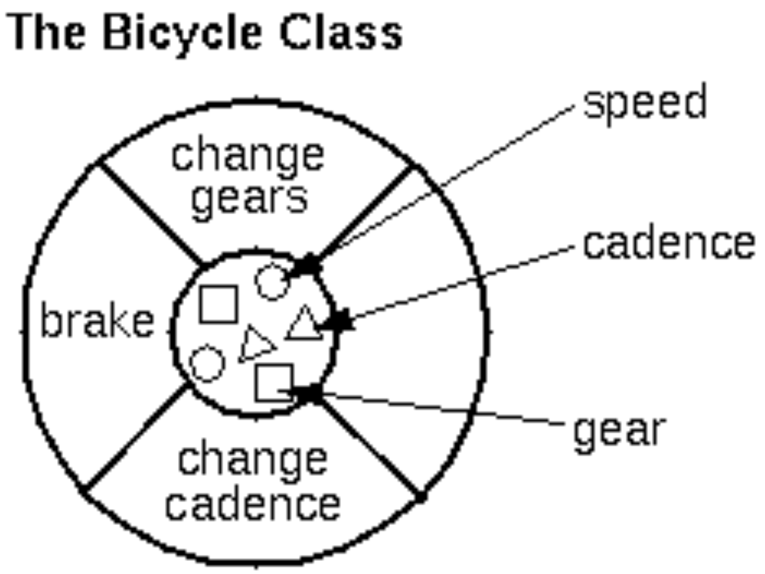
Syntax	Example	Output
Call a function that takes no arguments		
`<function_name>()`	`sys.exit()`	
Call a function that takes argument(s)		
`<function_name>(<arg1>, <arg2>)`	`len("Genomics")`	8

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

Syntax	Description	Example

`<str>.upper()`	• Returns the string with all letters uppercased	`>>> x = "Genomics"`
		`>>> x.upper()`

`<str>.lower()`	• Returns the string with all letters lowercased	`>>> x.lower()`
		`'genomics'`

`<str>.find(<pattern>)`	• Returns the first index of <pattern> in the string
	• Returns -1 if the if <pattern> is not found
		`>>> x.find('nom')`
		`2`

`<str>.count(<pattern>)`	• Returns the number of times <pattern> is found in the string
	• HINT: explore how .count deals with overlapping patterns
		`>>> x.count('g')`
		`0`

`<str>[<index>]`	• Returns the letter at the <index>th position	`>>> x[1]`
		`'e'`

0	1	2	3	4	5	6	7
G	e	n	o	m	i	c	s

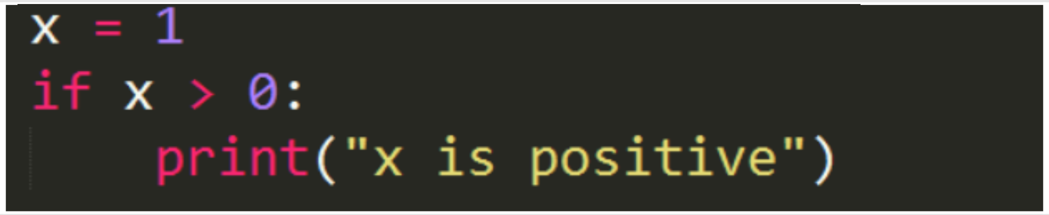
https://docs.python.org/3.4/library/stdtypes.html?#string-methods
https://docs.python.org/3/library/stdtypes.html#str

Making choices (conditional statements)

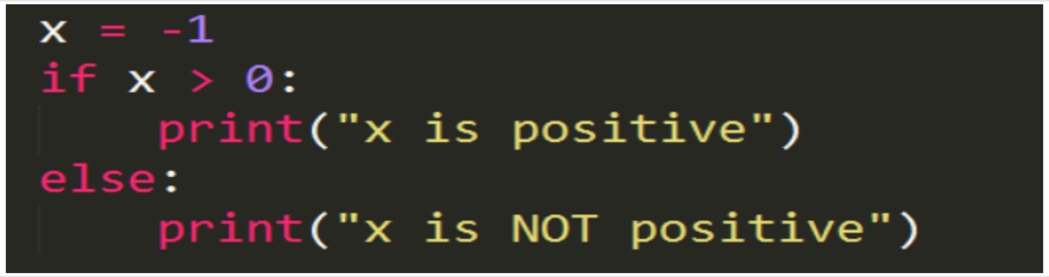
- Why is this concept useful?
  - Often we want to check if a condition is true and take one action if it is, and another action if the condition is false
  - *E.g., If the alternative allele read coverage at a particular location is high enough, annotate the position as a SNP otherwise, annotate the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

Conditional statement syntax

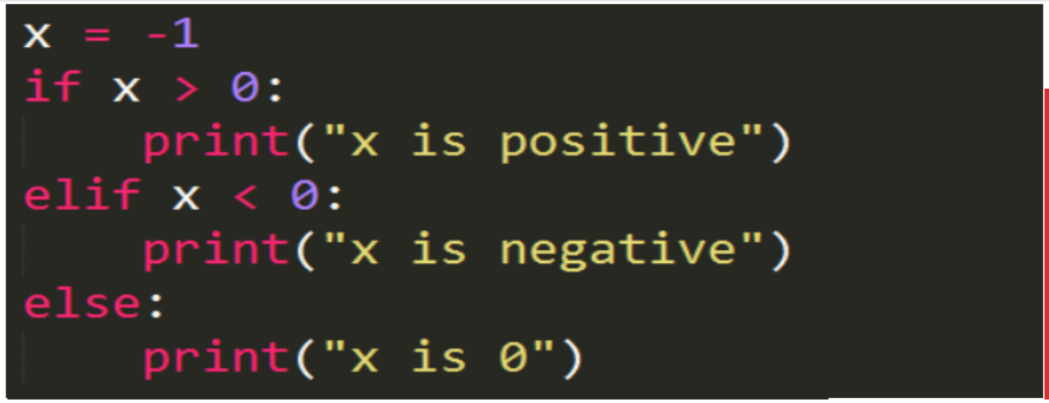
Syntax	Example	Output
If		

`if <condition>:`
`    # Do something`
	`x = 1`
	`if x > 0:`
	`    print("x is positive")`
		`x is positive`

If/else		

`if <condition>:`
`    # Do something`
`else:`
`    # Do something else`
	`x = -1`
	`if x > 0:`
	`    print("x is positive")`
	`else:`
	`    print("x is NOT positive")`
		`x is NOT positive`

If/else if/else		

`if <condition1>:`
`    # Do something`
`elif <condition2>:`
`    # Do something else`
`else:`
`    # Do something else`
	`x = -1`
	`if x > 0:`
	`    print("x is positive")`
	`elif x < 0:`
	`    print("x is negative")`
	`else:`
	`    print("x is 0")`
		`x is negative`
		
		Indentation matters!!!
		Indent the lines of code that belong to the same code block
		Use 1 tab

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

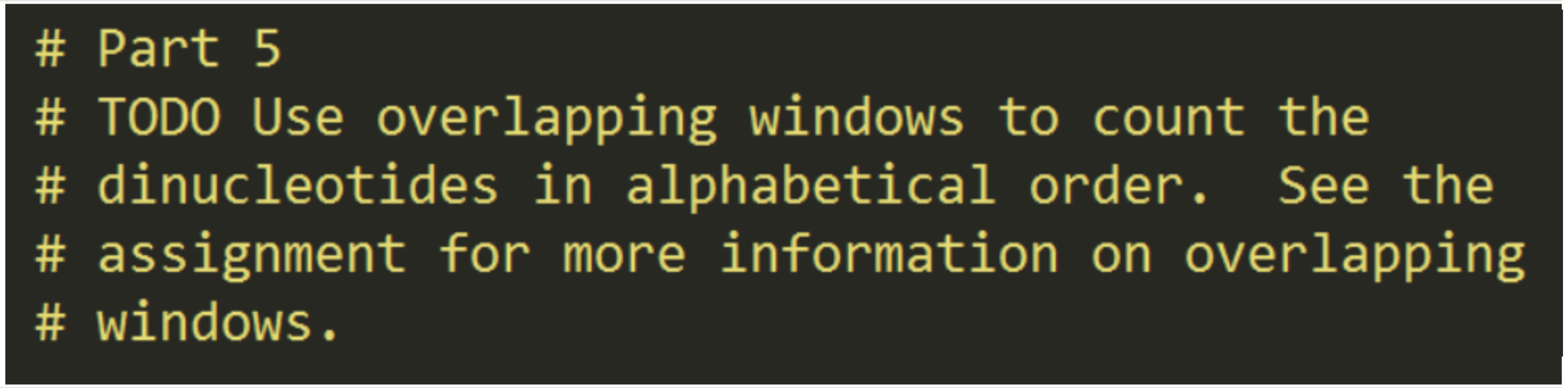
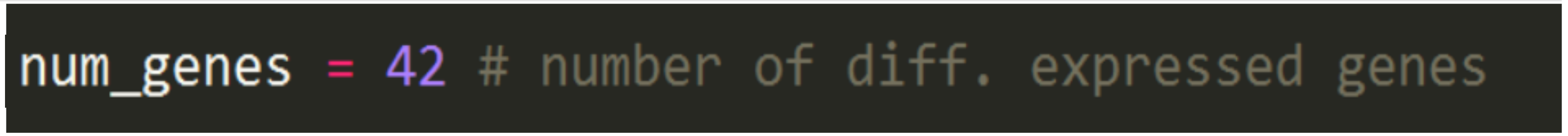
Always code [and comment] as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.

-- 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

Syntax	Example
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

Function	Description
`random.choice`	Return a random element from a list
`random.randint`	Return a random interger in a given range
`random.random`	Return a random float in the range [0, 1)
`Random.seed`	Initialize the (pseudo) random number generator

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

FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FRE
FOUR-LEAF CLOVERS ARE NOT M

For loop syntax

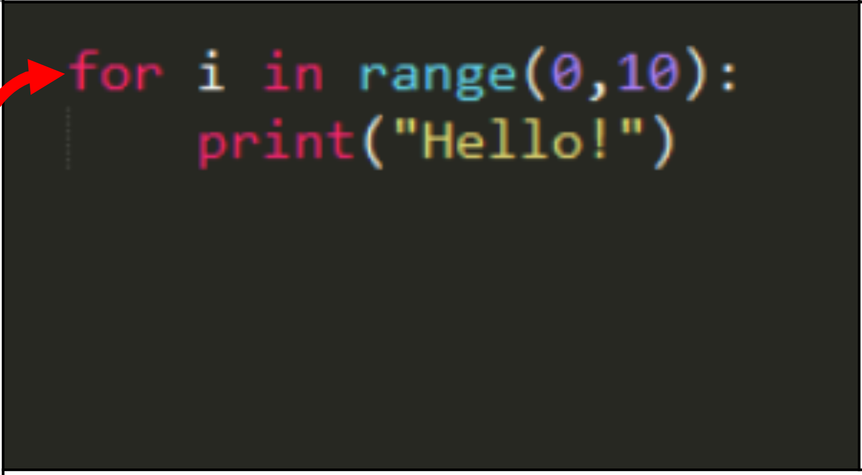
Indentation matters!!!
Indent the lines of code that belong to the same code block
Use 1 tab

Syntax

```
for <counter> in <collection_of_things>:
    # Do something
```

- 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!

Example

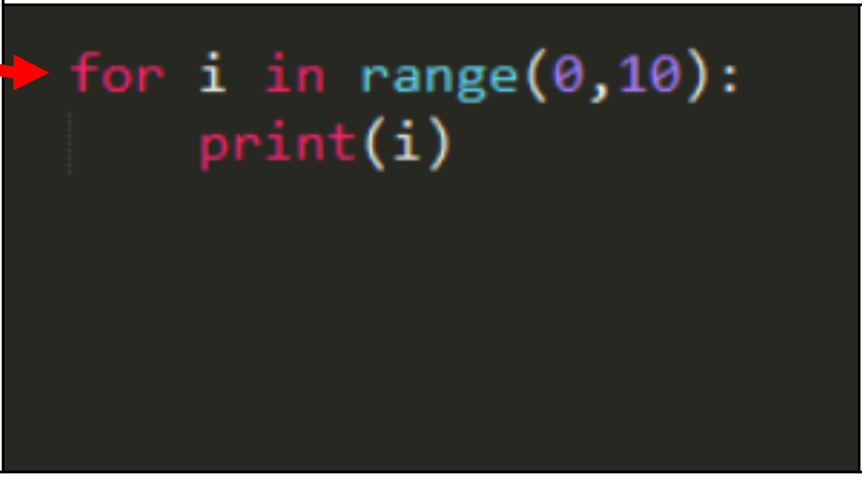


```
for i in range(0,10):
    print("Hello!")
```

Output

```
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
Hello!
```

Example



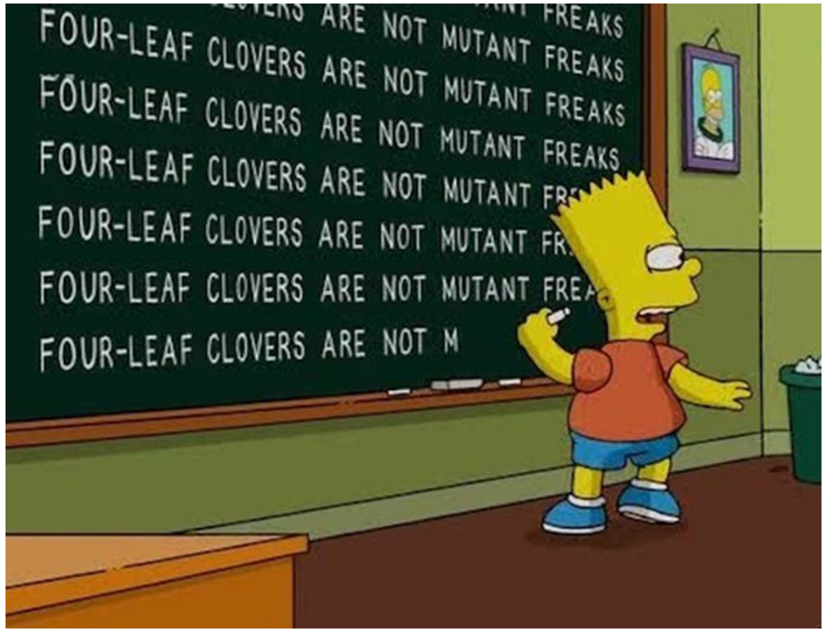
```
for i in range(0,10):
    print(i)
```

Output

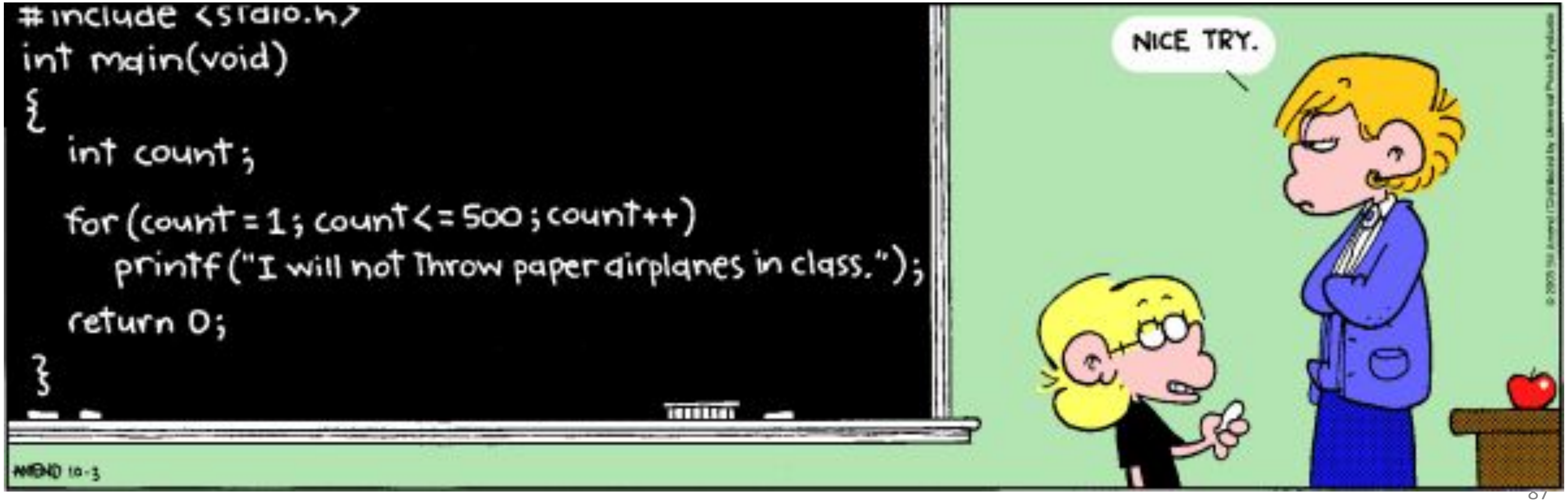
```
0
1
2
3
4
5
6
7
8
9
```

Which option would you rather do?

A


FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FREAKS
FOUR-LEAF CLOVERS ARE NOT MUTANT FR
FOUR-LEAF CLOVERS ARE NOT MUTANT FREA
FOUR-LEAF CLOVERS ARE NOT M

B



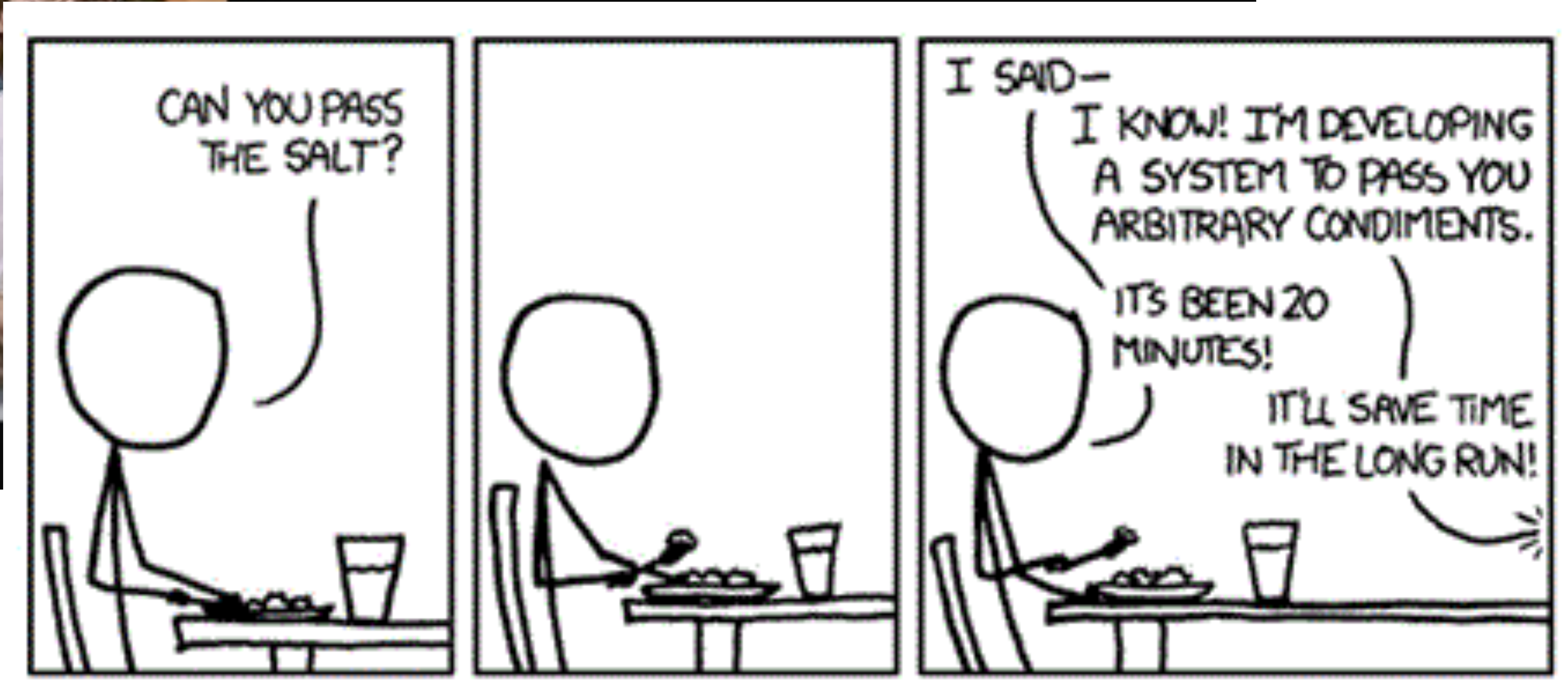
```
#include <stdio.h>
int main(void)
{
  int count;

  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 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

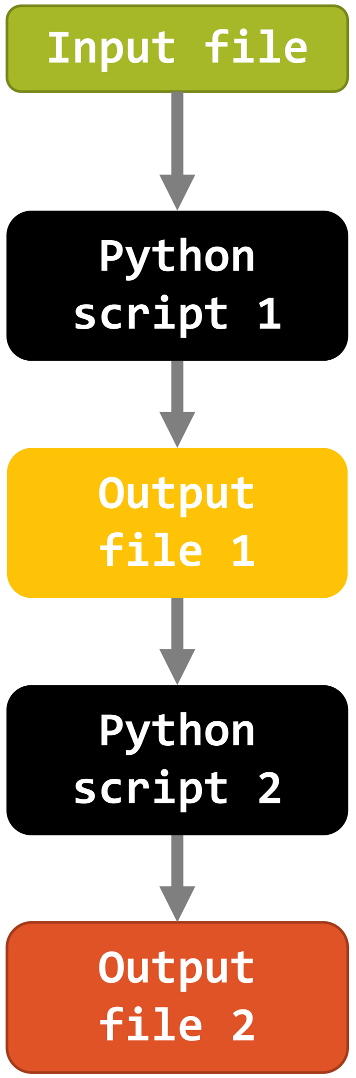
How do we solve this problem?

HOW'S THE BIG DATA PROJECT COMING ALONG, HOSKINS?
© D.Fletcher for CloudTweaks.com

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

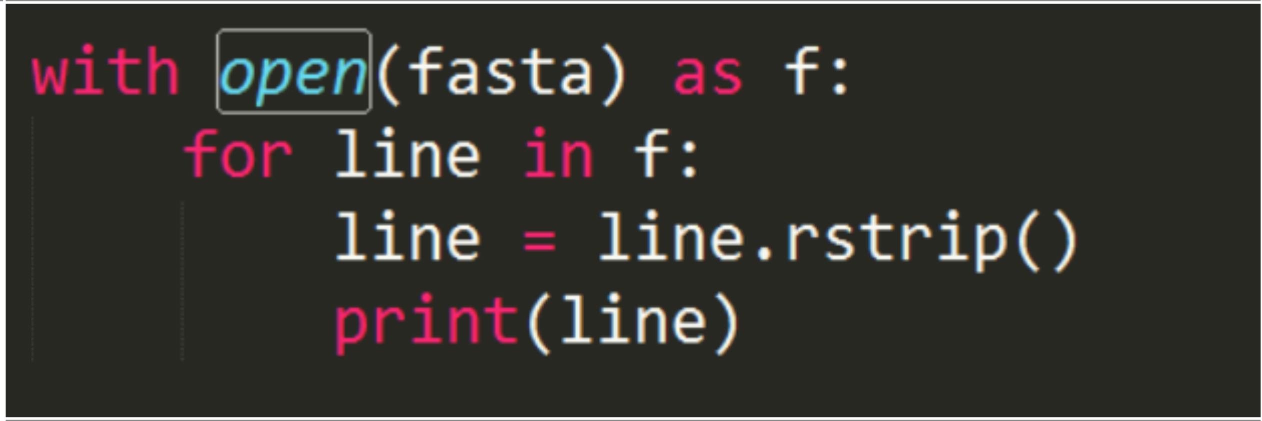


Reading a file syntax

Syntax

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

Example



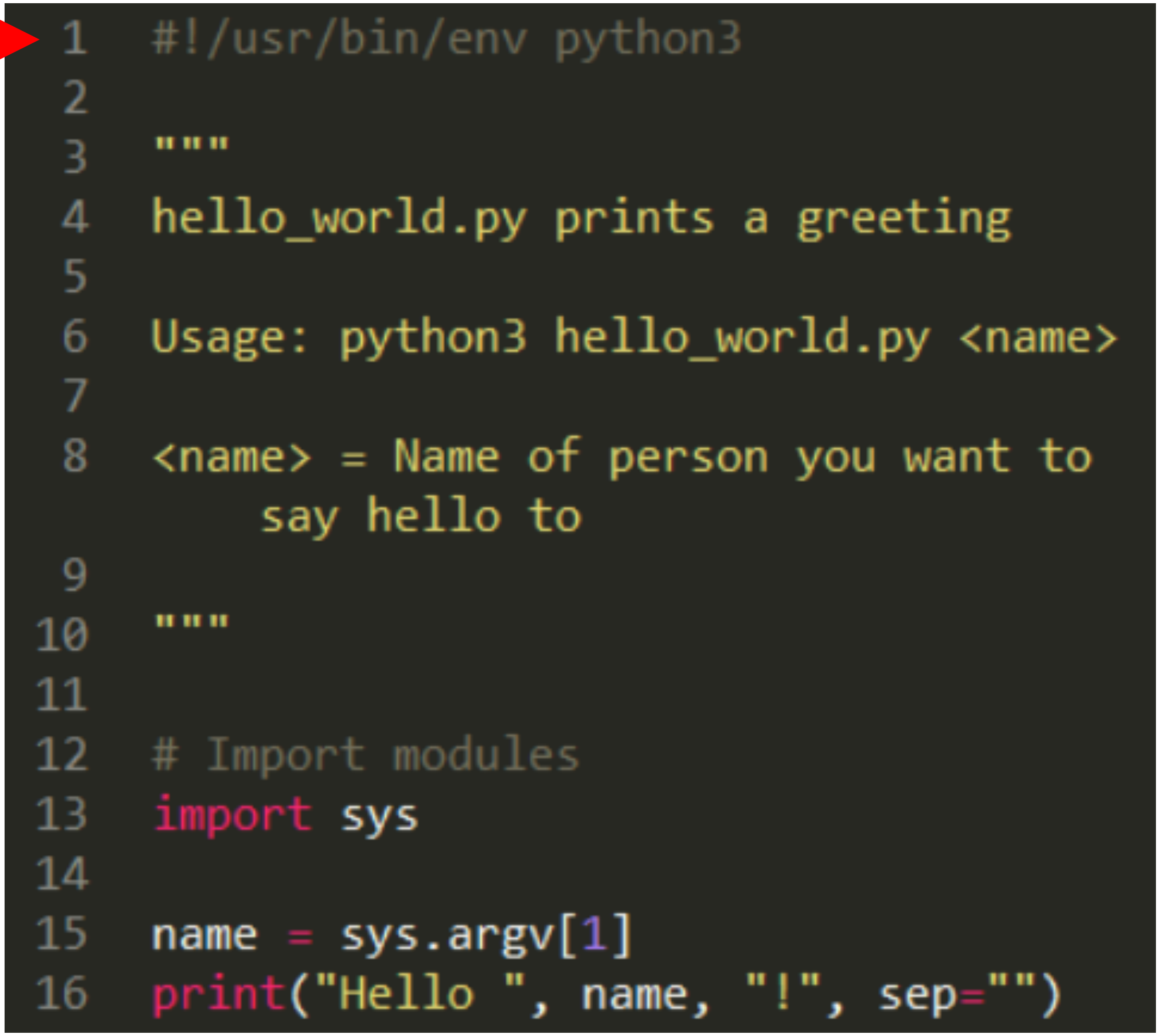
```
with open(fasta) as f:
    for line in f:
        line = line.rstrip()
        print(line)
```

Output

```
>chr1
ACGTTGAT
ACGTA
```

The anatomy of a (simple) script

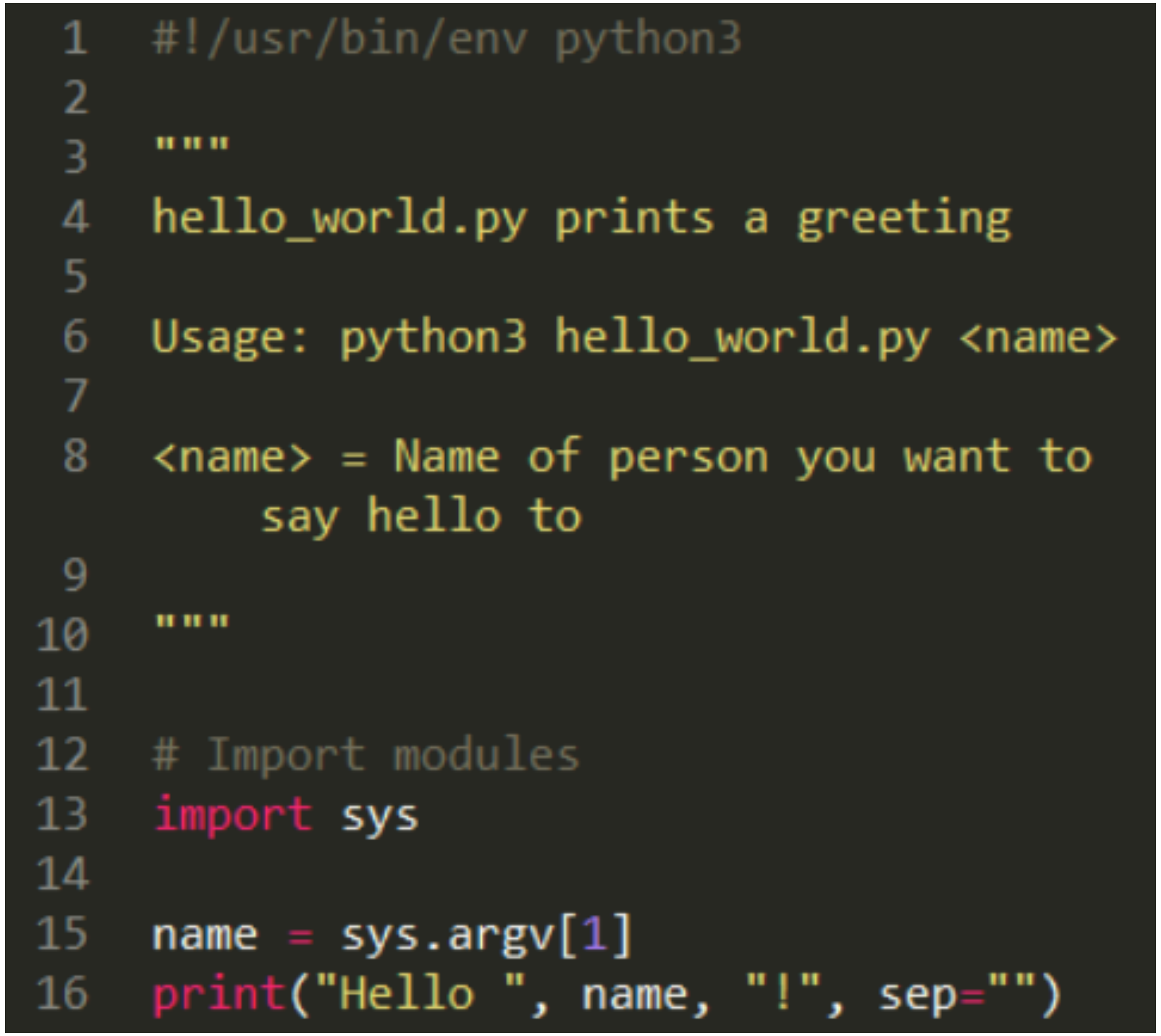
- The first line should always be `#!/usr/bin/env python3`
- This special line is called a **shebang**
- The shebang tells the computer how to run the script
- It is NOT a comment



```
#!/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

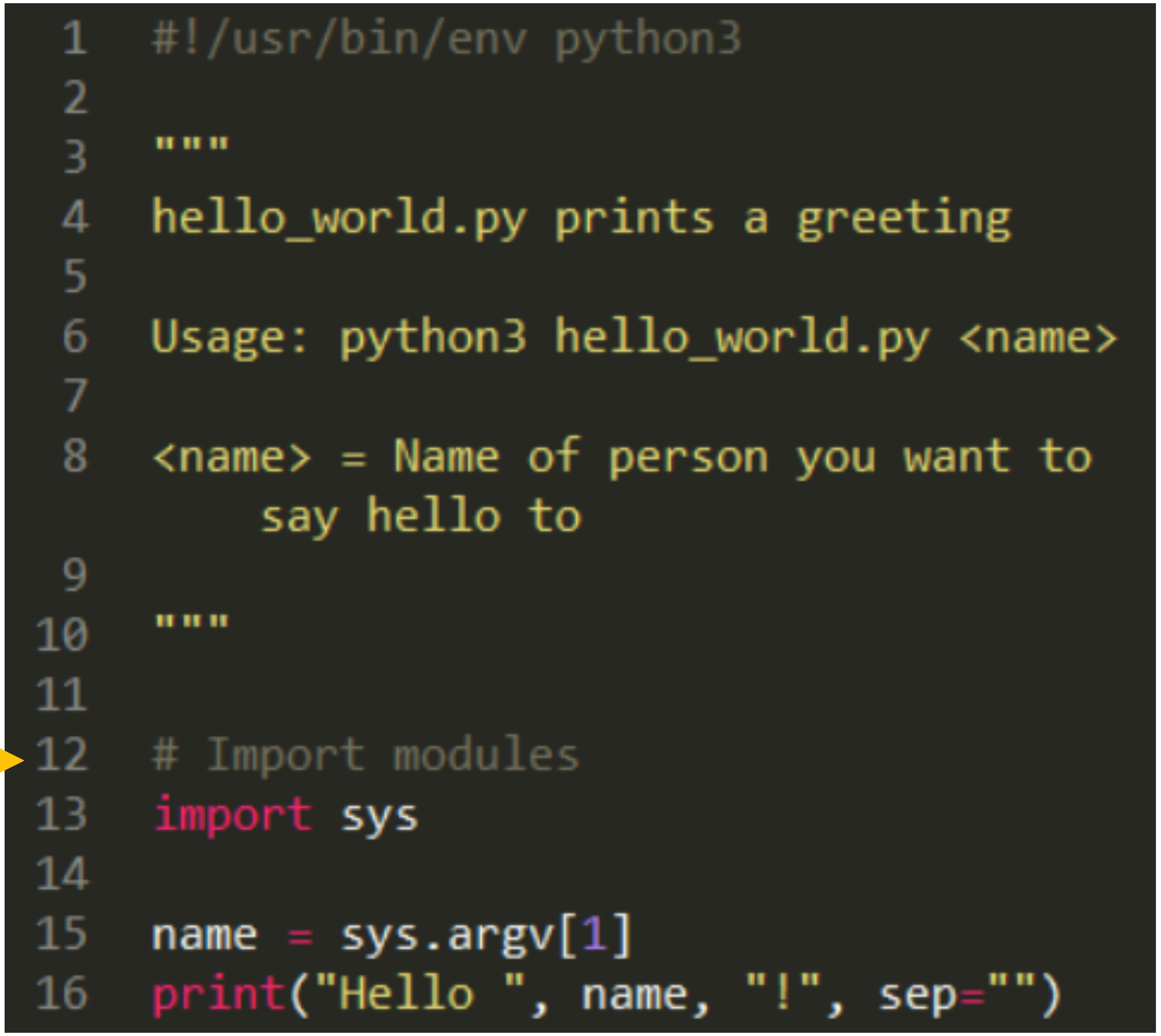
- This is a special type of comment called a **doc string**, or documentation string
- Doc strings are used to explain 1) what script does and 2) how to run it
- ALWAYS include a doc string
- Doc strings are enclosed in triple quotes, """



```
#!/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="")
```

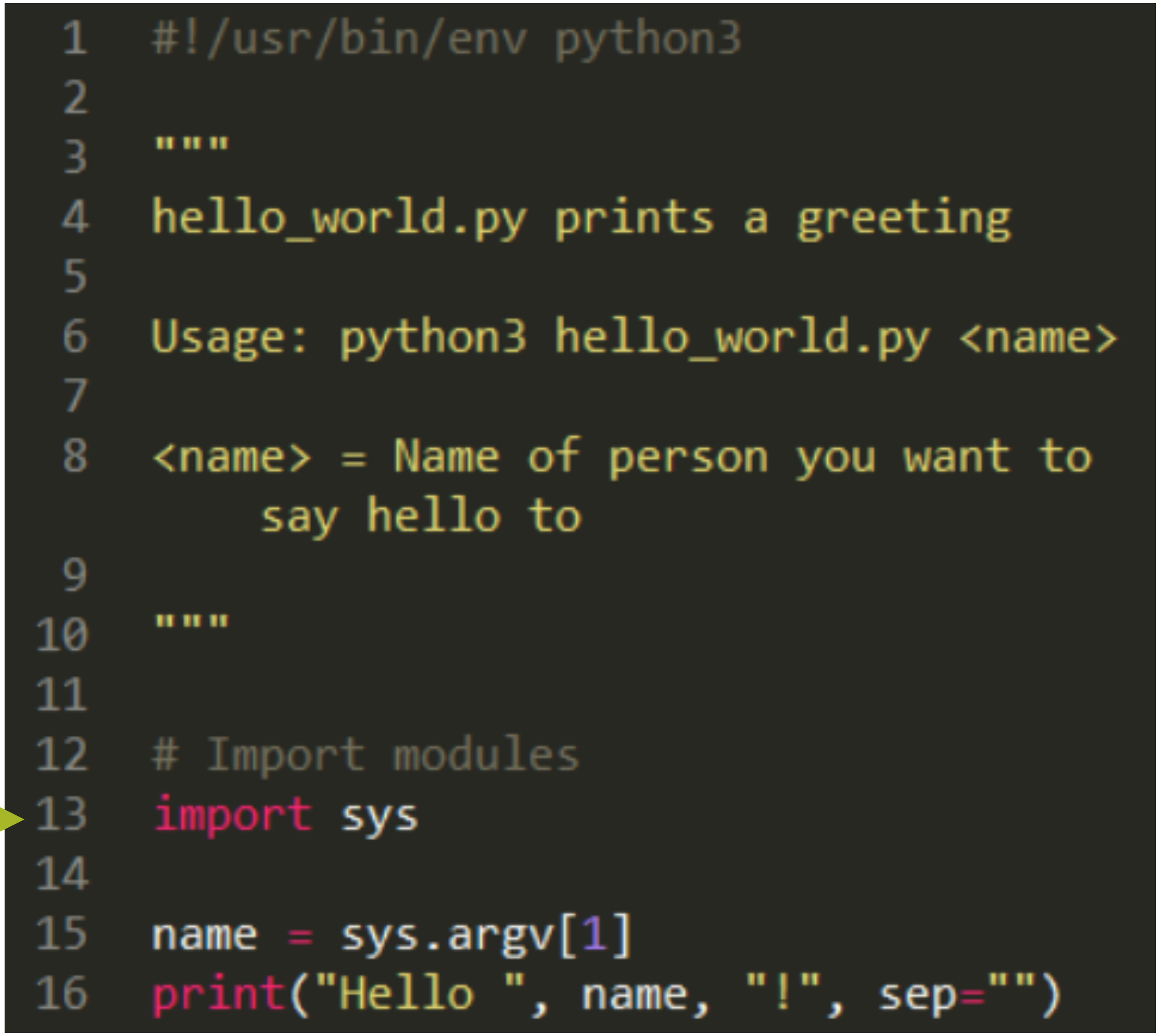
The anatomy of a (simple) script

- This is a comment
- Comments help the reader better understand the code
- Always comment your code!



```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="")
```

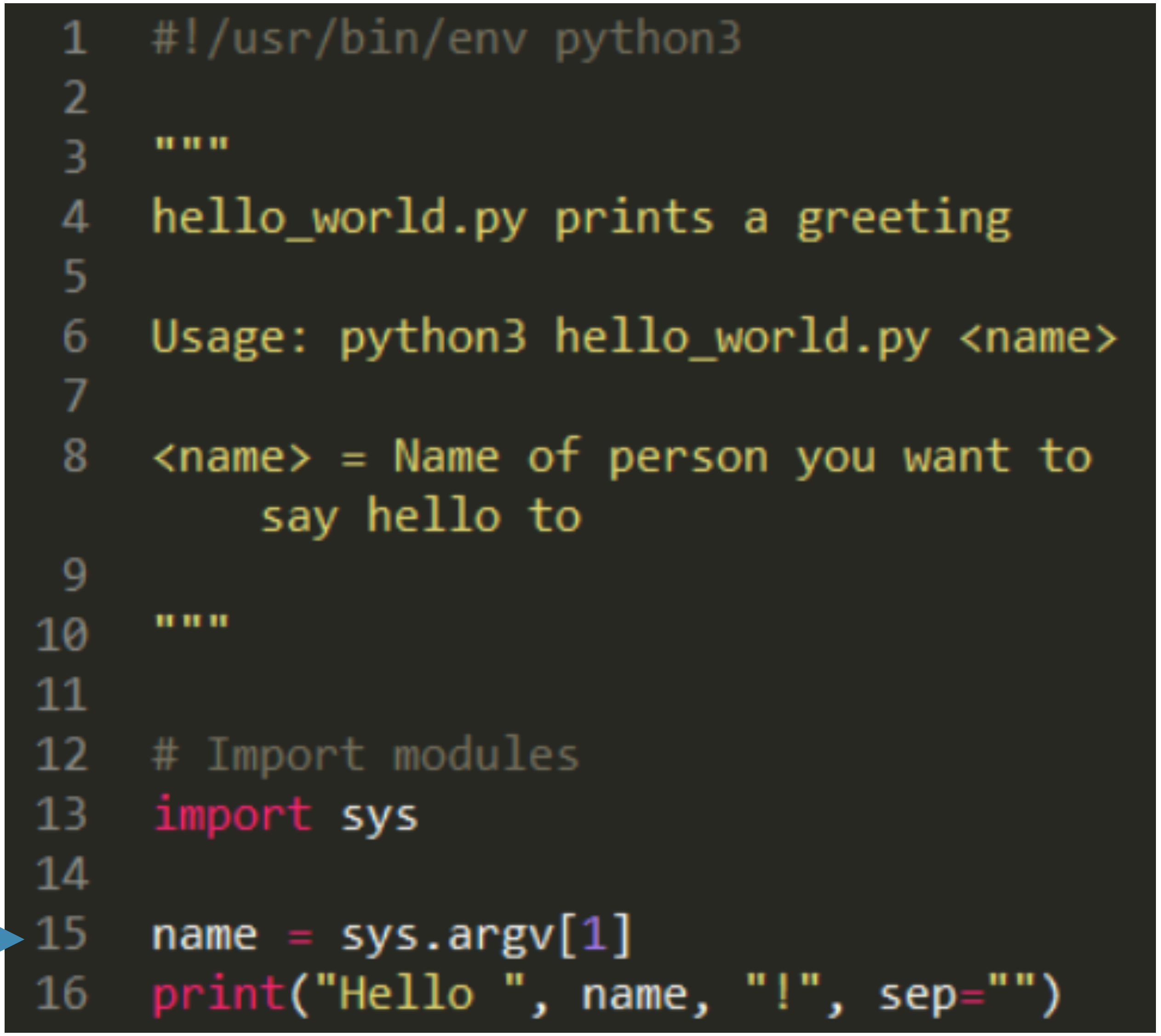
The anatomy of a (simple) script



```
#!/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="")
```

- 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

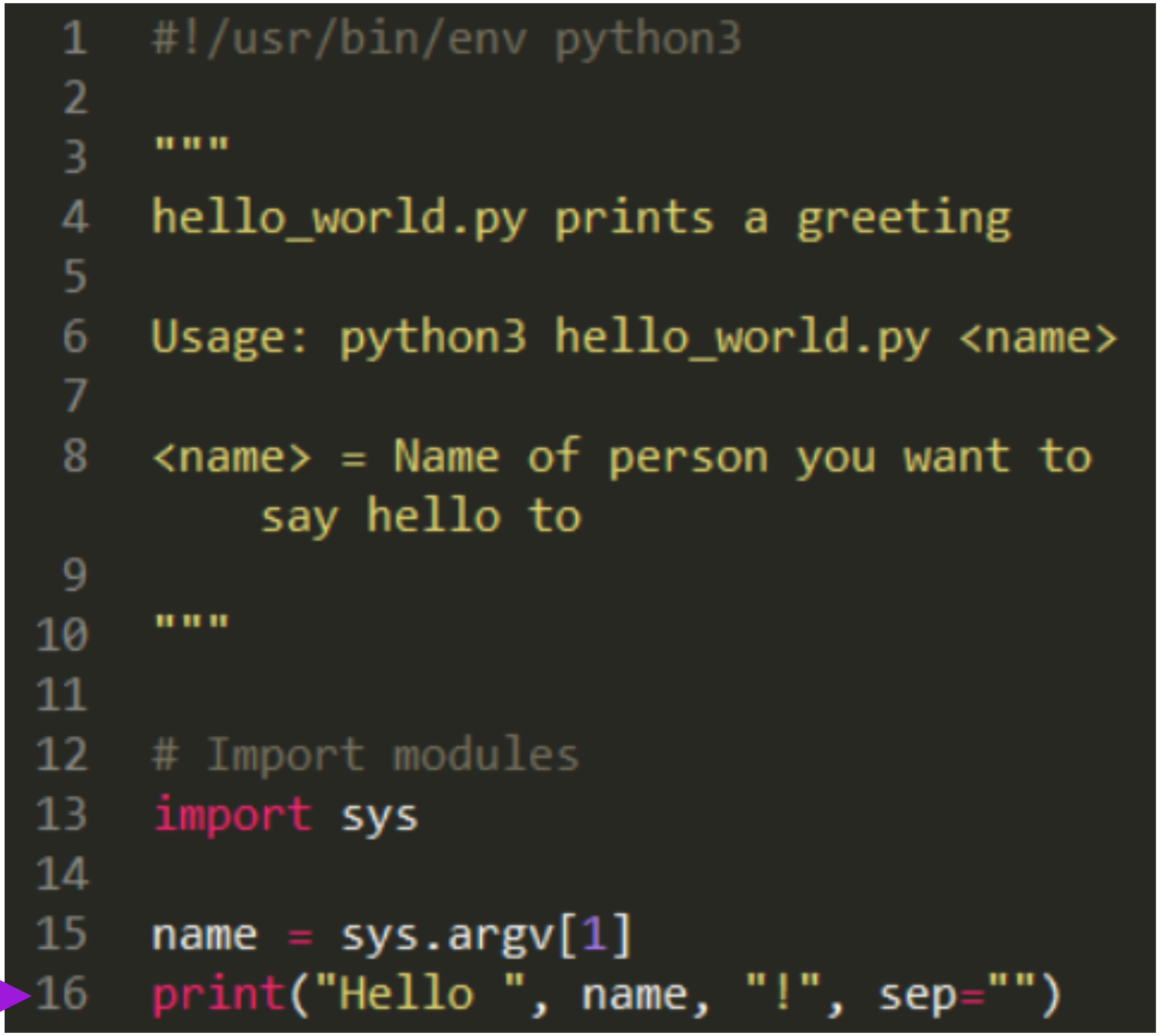
The anatomy of a (simple) script



```
#!/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="")
```

- This grabs the command line argument using `sys.argv` and stores it in a variable called name

The anatomy of a (simple) script



```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="")
```

- 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.

KEEP
CALM
AND
READY
TO GO