# 1 Introduction

## 1.1 Learning objectives

This set of prepwork chapters is intended to give you a brief exposure to the fundamentals of the Python programming language. By the time you’ve completed the prepwork, you should be able to

* Navigate around a file system using the command line
* Understand why we use both bash and Python in Computational Biology to analyze biological data
* Understand the basic data types in Python
* Open file streams
* Use conditionals
* Use a for-loop to iterate
* Write print statements
* Use pseudo-coding to identify problems/steps to a solution

## 1.2 Motivation

We recognize that each of you will be joining the CMDB program with your own unique background. This means that prior exposure to coding will vary from never having used the command line to proficient coders.

For those of you to whom this is all new, know that we understand that this will be challenging but we believe that each and every one of you are capable of learning to code and become capable at bioinformatics. If you feel lost, ask for help. Our goal is for every person in this class to get an A, but more importantly to feel confident in your ability to handle basic programming and bioinformatic tasks.

For the people already comfortable with programming, there are still things to learn, especially with respect to applying your knowledge to bioinformatic problems. Sometimes you will hear things you already know. Just take it as review. But more importantly, take it as an opportunity to help pass on that experience to your fellow classmates.

You all are a cohort of graduate students. The best way to succeed, not just in this class but in your graduate career is to form a support structure with your fellow students now. Helping each other succeed will make your time here more fruitful and enjoyable. You are all in this together and we are here to facilitate your learning, provide support, and encourage.

## 1.3 Assistance

If at any time you encounter difficulty, please feel free to reach out to one of the TAs through Slack or email. This includes if you have questions or get an error while running through exercises in any chapter and need help debugging.

* Kate Weaver [kweave23@jhu.edu](mailto:kweave23@jhu.edu)
* Stephanie (Steph) Yan [syan11@jhu.edu](mailto:syan11@jhu.edu)
* Dylan Taylor [dtaylo95@jhu.edu](mailto:dtaylo95@jhu.edu)
* Andrew Bortvin [abortvi2@jhu.edu](mailto:abortvi2@jhu.edu)

# 2 Logistics

## 2.1 Goal

Below we describe two ways you can complete these exercises:

* Online Python – Work “on the Cloud” using your laptop, browser, and internet connection
* Your Own Computer – Work “locally” with your own Python and text editor

By the end of the actual course, you will be able to run Python scripts from whatever machine you are working on, including accessing local files

## 2.2 Macs

If you are using a Mac to complete the prepwork, you should be in good shape. Python should be already installed on your computer. You will also need a plain text editor such as TextEdit, TextMate, Sublime, etc. (NOT a wordprocessor like Word, Pages, etc.). Finally, you will need to open iTerm or another comparable terminal program. This will bring up a command line terminal. You can verify that Python is installed by running the command which python on the command line. If the result is a filepath ending with some version of ‘python’, you are all set.

If for some reason Python is not installed or you would prefer to work online, jump to the [Online Python](#Online-Python) section.

## 2.3 Windows

Unfortunately, Python does not come pre-installed on the Windows OS. You have a couple of options. First, you can install it by downloading the installer from [here](https://www.python.org/downloads/windows/). Select whatever Python3 version you want (I would go with 3.10). Then follow the instructions when you run the downloaded installer. Finally, verify that Python was successfully installed by opening Command Prompt and running the command python -V. This should tell you what version of Python is installed on your system. In addition to Python and Command Prompt, you will also need a plain text editor like Notepad, Sublime, etc.

If you would prefer not to install Python, jump to the [Online Python](#Online-Python) section.

## 2.4 Unix

If you are on a Unix system, Python should be pre-installed. If not, chances are that you are comfortable installing it yourself using apt-get, dnf, or whatever the appropriate package manager is for your distro. You will also need a terminal and plain text editor.

If for some reason Python is not installed or you would prefer to work online, jump to the [Online Python](#Online-Python) section.

## 2.5 Online Python

During your first year in the CMDB program, you will be provided with a laptop with everything you will need preloaded on it. However, for the prepwork you can make use of a Python interpreter available online [here](https://www.online-python.com). If you are not familiar with navigating your filesystem from the command line, we highly recommend making use of the online interface. The online interface includes both a text editing interface to write your scripts in and save them using the “disk” icon above the editor. Below the editor is a “Run” button that executes your code, printing any output in the window below the editor. You can also load external files to make them accessible to your script with the “folder” icon above the editor. Finally, there is a text field that you can use to pass arguments to your script.

## 2.6 Submitting your final script

Once you have completed the last module, you will need to submit the script you have produced (the tail script). To do this, please name it LASTNAME\_tail.py and email it to Kate Weaver at [kweave23@jhu.edu](mailto:kweave23@jhu.edu).

# 3 Bash

## 3.1 Goal

Gain an understanding of what bash is and what it can do for you

## 3.2 Learning Objectives

After going through this module, students should be able to:

* Explain the difference between the command line and bash
* List advantages of bash over a graphical user interface (GUI)

## 3.3 What is bash

In order to understand what bash is, we first need to define what the command line is. The command line is the text interface with your computer that is brought up when you run terminal or another similar program. In computers with no graphical interface, the command line is the only way to interact with the computer.

Bash is a command line interpreter. There are several different interpreters, but bash is quite common and the default on the provided CMDB laptops. Bash allows you to:

1. navigate through the filesystem
2. interact with files by creating, editing, examining, moving, or deleting them
3. run built-in tools
4. perform some basic logical operations, i.e. a simple form of programming

Though it lacks some of the features of other programming languages (which we will discuss in the Chapter 5), bash is similar in that it executes commands. Bash is particularly specialized for file system navigation and management, networking, and administration tasks, such as setting up the environment (what programs are available to be used), and chaining together code written in other languages.

## 3.4 Bash vs. a graphical user interface (GUI)

While all of us are very familiar with GUIs (e.g. macOSX or windows, excel, gimp, photoshop), many will be less (or not) familiar with getting the same or more functionality out of the command line through bash. Some of the most powerful features of using bash are text manipulation and performing repetitive tasks. While on a GUI there is not an easy way to perform the same task or set of tasks repeatedly, bash allows you to create a script that can easily handle even complex sets of tasks repeated on each input.

The biggest disadvantage of the command line and bash compared to a GUI is the learning curve. Many people are nervous to use the command line because of its unfamiliarity, but with a few simple commands, you will be using bash with ease.

# 4 Basic bash

## 4.1 Goal

A basic grasp of navigating bash and using some basic commands

## 4.2 Learning Objectives

After going through this module, students should be able to:

* Move between directories in bash
* Examine folder contents
* Move and copy files
* Run the head and tail commands
* Describe what a usage statement is and print it for head and tail
* Explain what a “man page” is

## 4.3 Resources

* [Bash syntax cheatsheet](http://bxlab.github.io/cmdb-bootcamp/cheatsheet/unix.html)

This module will be performed online in a [bash sandbox](https://sandbox.bio/tutorials?id=terminal-basics). This avoids issues that can arise from people using different operating systems and having trouble finding the terminal on their own computers. Once you open the link, you should see a black window with the prompt “[guest@sandbox](mailto:guest@sandbox)$”. This is the bash interface.

## 4.4 Navigation

When you use bash, your system is located or working within a single folder (which we refer to as a directory) at any given time. Whatever directory you are in is known as the “working directory”. You can always see what directory you are working in with the command pwd, which stands for “print working directory”. To run the command, simply type “pwd” into the terminal window and press <Enter> or <return>.

What directory name was printed out?

ANSWER:

/shared/data/tutorials/terminal-basics

You can see that not only the name of the working directory was printed, but also, the name of every parent directory all the way up to the **root**, the left-hand “/” that contains all files and directories on the computer. This is nested list of directories is called the “path”. In this case, it is an **absolute** path because it tells us how to get to this directory from the root.

In order to move to another directory, we can use the command cd, which stands for “change directory”. If we wanted to move to the “tutorials” directory, we could type the cd command and the path to that directory.

cd /shared/data/tutorials

Once you have done this, verify that you are in this directory using the pwd command. In this case, “tutorials” is the **parent** directory of “terminal-basics”.

If we had to type a long directory path in every time we wanted to change directories, it would be very tedious, especially with highly nested directories. Luckily, there is another way to navigate using **relative** paths. This means giving directions based on your current working directory. So, to move back to the “terminal-basics” directory, instead of typing out the full path, we will simply give the directory name since it is located in our working directory.

cd terminal-basics

But what about going back to the “tutorials” directory? To do that, we need to use the special notation “..” which means the parent of a directory. So, to go back to the “tutorials” directory, use the following command.

cd ..

Use pwd to make sure that you are working in the “tutorials” directory now. If you are in the “tutorials” directory, you don’t need to do anything until the paragraph “You can also give longer relative …”. If you aren’t in the “tutorials” directory, review what commands you used by pressing the up arrow and compare your history of commands to the directions above. Pressing the up arrow will show you a command you used, most recent first. Pressing it again will show you the command you used before that, continuing all the way to the first command being the last time pressing the up arrow changes what command is displayed on the screen. Pressing the down arrow will take you back through down the history back to the most recent command. Then use an absolute path to move back into the “tutorials” directory: cd /shared/data/tutorials

cd /shared/data/tutorials

You can also give longer relative paths by combining directions. The directory “shared” which contains “data” also contains a directory named “home”. To move there from the “tutorials” directory, use the relative path “../../home”.

cd ../../home

Now, using a relative path, how can you get back from /shared/home to the directory where we began this lesson /shared/data/tutorials/terminal-basics?

ANSWER:

cd ../data/tutorials/terminal-basics

Finally, one other special notation that you can use is **~** which stands for your home directory. You can always use cd ~ to return to your home directory. You can also use relative paths from your home directory, like cd ~/mystuff, assuming that the “mystuff” directory exists. Home directories are computer specific, so we will discuss these more in the actual course when you have the CMDB provided laptop.

## 4.5 Examining directory contents

Navigating around the filesystem is only useful if we can also see what is in each directory. To do this, we use the command ls which stands for “list”. Make sure you are in the “terminal-basics” directory, then type “ls” and press enter. What is the command output (i.e. the text printed in the terminal)?

ANSWER:

orders.tsv ref.fa ref.fa.bak

You should see three files in this directory. If there were any directories here, they would also be listed but there would be no way to tell the difference from the file names. To do that, we need more detail, which is provided by using the option “-l”, which stands for “long”. Let’s see what extra detail is provided.

ls -l

total 14  
-rw-rw-rw- 1 0 0 105054 Aug 3 10:53 orders.tsv  
-rw-rw-rw- 1 0 0 45 Aug 3 10:53 ref.fa  
-rw-rw-rw- 1 0 0 45 Aug 3 10:53 ref.fa.bak

The ls -l command tells us about file permissions and several other attributes (don’t worry about these for now), the file size in bytes, the latest modification date, and the name. If you want easier to read file sizes, add the option “-h”, which will print a “human readable” size format.

We can also use ls with a specific target, for example another directory. If we run the command on our parent directory, we can see what a directory looks like.

ls -l ..

total 1  
drwxrwxrwx 1 0 0 4096 Aug 3 10:53 terminal-basics

The “d” at the left-hand side indicates that this is a directory, not a file.

## 4.6 Moving and copying

What if we need to make a copy of a file or directory because we want to change something but keep a backup? To do this, we can use the cp command, short for copy. The command takes a target, the file you want to copy, and a destination. The destination can either be a directory (in which case the copied file will have the same name, but be located in a new additional location), or a new name (with or without a path). Unfortunately, the online terminal does not allow copying files, so do not type the following command, but if we wanted to copy the “ref.fa” file to “ref.fa.backup”, we would do the following within the “terminal-basics” directory.

cp ref.fa ref.fa.backup

Then, if we were able to copy the file follow copying with the ls command, you would see an additional file in the directory.

Copying a directory is similar but because directories can contain other things that need to be copied as well, you need to include the option “-R” for recursive. Unfortunately, the online terminal also does not allow copying directories, so do not type any of the following commands. If the online terminal did allow copying directories, it would look something like this:

ls -l

total 1  
drwxrwxrwx 1 0 0 4096 Aug 3 12:30 terminal-basics

cp -R terminal-basics new-basics  
ls -l

total 2  
drwxrwxrwx 1 0 0 4096 Aug 3 12:30 terminal-basics  
drwxrwxrwx 1 0 0 4096 Aug 3 12:31 new-basics

We can also move and rename files using the mv command. Like copy, you give it a target file or directory and a destination. If the destination is a directory, the target is moved into that destination directory. If the destination is a file name (with or without a path), then the target is renamed (and moved if there is a path to a different directory than the target is in).

Unfortunately, the online terminal also does not allow moving but it does allow renaming. So, if we are in “terminal-basics” and want to rename our “ref.fa.bak” file to “my\_ref.fa”, what command would we use? What about moving it into the parent directory? And if we wanted to move it and give it the new name “my\_ref.fa”?

ANSWER for renaming ref.fa.bak to my\_ref.fa:

mv ref.fa.bak my\_ref.fa

ANSWER for moving ref.fa.bak to the parent directory:

mv ref.fa.bak ..

ANSWER for moving it to the parent directory with a new name my\_ref.fa:

mv ref.fa.bak ../my\_ref.fa

## 4.7 Head and tail

There are many bash commands besides those we’ve seen: pwd, cd, ls, cp, and mv. Right now we will focus on two more, head and tail. The head command reads through a text file, printing out the first *n* lines, where *n* is an arbitrary number passed as an option with -n. If *n* is not specified, head defaults to 10 lines. tail is similar but instead of printing out the first *n* lines, it prints out the **last** *n* lines. Try both of these out on the file “orders.tsv”.

head orders.tsv

What do you get for output from head?

ANSWER:

id num item\_name  
1 1 Chips and Fresh Tomato Salsa  
1 1 Izze  
1 1 Nantucket Nectar  
1 1 Chips and Tomatillo-Green Chili Salsa  
2 2 Chicken Bowl  
3 1 Chicken Bowl  
3 1 Side of Chips  
4 1 Steak Burrito  
4 1 Steak Soft Tacos

tail orders.tsv

What do you get for output from tail?

ANSWER:

1831 1 Carnitas Bowl  
1831 1 Chips  
1831 1 Bottled Water  
1832 1 Chicken Soft Tacos  
1832 1 Chips and Guacamole  
1833 1 Steak Burrito  
1833 1 Steak Burrito  
1834 1 Chicken Salad Bowl  
1834 1 Chicken Salad Bowl  
1834 1 Chicken Salad Bowl

If you were to forget the default number of lines or need more information about head, you can get more information from the *command help*, a detailed list of options and arguments and the usage statement. To see the command help for head, use the option “–help”.

head --help

Usage: head [OPTION]... [FILE]...  
Print the first 10 lines of each FILE to standard output.  
With more than one FILE, precede each with a header giving the file name.  
  
With no FILE, or when FILE is -, read standard input.  
  
Mandatory arguments to long options are mandatory for short options too.  
 -c, --bytes=[-]NUM print the first NUM bytes of each file;  
 with the leading '-', print all but the last  
 NUM bytes of each file  
 -n, --lines=[-]NUM print the first NUM lines instead of the first 10;  
 with the leading '-', print all but the last  
 NUM lines of each file  
 -q, --quiet, --silent never print headers giving file names  
 -v, --verbose always print headers giving file names  
 -z, --zero-terminated line delimiter is NUL, not newline  
 --help display this help and exit  
 --version output version information and exit  
  
NUM may have a multiplier suffix:  
b 512, kB 1000, K 1024, MB 1000\*1000, M 1024\*1024,  
GB 1000\*1000\*1000, G 1024\*1024\*1024, and so on for T, P, E, Z, Y.  
Binary prefixes can be used, too: KiB=K, MiB=M, and so on.  
  
GNU coreutils online help: <https://www.gnu.org/software/coreutils/>  
Full documentation <https://www.gnu.org/software/coreutils/head>  
or available locally via: info '(coreutils) head invocation'

The first line shows the usage statement, a description of how to run the command. Options and arguments (information passed without a preceeding option identifier like -n) are denoted as optional when enclosed by **[]** and required when enclosed by **<>** or nothing at all. The usage statement also provides the order of options and arguments required by the command.

Most commands and tools have a help statement and usage statement, but not all. In addition, we can get even more information by looking at the manual, or **man** page for a command. To do this, we would type man head. Unfortunately this is one more feature missing in this online terminal. A man page is a text manual with a general description of the tool, an explanation of all of the possible arguments, and more. You can navigate with the up and down arrows. To move more quickly, hold down *fn*. To exit, push *q*.

# 5 Python

## 5.1 Goal

Get a basic familiarity with what kind of language Python is and why we’re using it

## 5.2 Learning Objectives

After going through this module, students should be able to:

* List the advantages and limitations of Python as a programming language
* Explain why Python is a complement to bash
* Understand why Python is being used for this class

## 5.3 What Python is

Python is a popular language for many applications, including bioinformatics. That is because it is easy to learn and read. In addition, it is well documented and supported by a large community. The are a vast number of packages extending Python’s functionality, including many specifically geared towards biological and bioinformatic applications.

Python is a high-level programming language. What that means is that much of fiddly details needed to make your code work is hidden under the hood and you will never need to worry about them. This makes writing code simple and fast. In addition, Python does not need to be compiled. This allows you to execute code as you write it.

Finally, Python is an object-oriented programming language meaning that every variable and function in python is actually a class that has built-in attributes and functions. While this may not make sense yet, it makes Python a very powerful language.

## 5.4 What Python is not

Although it is a clear language to read and write, Python is not a miracle cure for programming. You will still need to learn the syntax of the language, as well as it’s relatively small base vocabulary. Finally, you will still need to be able to translate strategies for answering questions or performing tasks into a logical set of steps that can be executed.

Python is also slower than compiled langauges and bash. This is especially true when dealing with loops. However, there are ways to significantly speed up the execution of Python. Many python packages have compiled C-code hidden within their function calls, such that you have the convenience of Python but the speed of a compiled language.

## 5.5 Why do we need Python if we have bash?

While bash has many powerful functions and basic programming logic, it is not a true programming language but rather a command-line interpreter. There are several advantages and some disadvantages to using Python over bash.

| Bash | Python |
| --- | --- |
| Difficult to write | Easy to write |
| Hard to read | Easy to read |
| Less powerful | More powerful |
| Faster | Slower |
| No third-party programs needed | Third-party programs may be needed |

Ultimately, you should be using these two approaches together to take advantage of the strengths of each. You will find that creating an analysis pipeline is quite effective when using bash to connect the inputs and outputs of multiple programs (including your own Python scripts) to take raw data through processing and analysis.

## 5.6 Why Python is right for this class

Given the range of student backgrounds and exposure to coding, Python offers a useful middle ground of easy to read and learn code while maintaining the power and flexibility that more advanced coders may need.

If you perform any data analysis in biology (and you will), it is inevitable that you will encounter programs written in Python. You may never see the code, but exposure to programming concepts now will give you a greater understanding of

1. how the programs work
2. troubleshooting issues when running them
3. tackling common problems like reformatting data

Most importantly, writing code in Python should be fun, not scary. And that is what will keep you using the skills that you learn in this class.

# 6 Pseudo-code

## 6.1 Goal

Understanding how to break a problem down into basic steps that can be translated into code

## 6.2 Learning Objectives

After going through this module, students should be able to:

* Explain the purpose of pseudocode
* Dissect a problem into basic building blocks
* Verbalize the goal of each step

## 6.3 What is pseudocoding?

Pseudocoding is a way of informally describing the logic behind solving a potentially complicated task by breaking it down into individual steps and explaing the steps in plain language.

An example of pseudocode for the task of assigning pass/fail grades to students in a class:

SET students ## Add students to a variable  
SET grades ## Add student grades to a variable  
INITIALIZE outcomes ## We also need somewhere to store the results  
SET cutoff ## We need some passing criteria  
FOR each student in students ## We need to take the same action for each student  
 IF student's grade is greater than or equal to cutoff  
 then ADD 'pass' to outcomes ## Update the outcomes  
 OTHERWISE  
 add 'fail' to outcomes  
 END IF  
END FOR

You’ll notice that all of the code that is to be repeated for each student is indented to indicate what is included in the repeated block. The same is true under the conditional statement. This helps us keep track of what steps constitute a subtask when code is repeated or run only under certain conditions. There is also the challenge of matching up the grade with the student. This can be accomplished in many different ways and will vary with what language is being used. Here, we keep it language-agnostic.

## 6.4 Why pseudocode instead of just writing a script?

The advantage of writing pseudocode first rather than just writing a script in your programming language of choice is that it is

* Helps transition novice programmers into thinking through the logic needed to solve a task
* Easier to breakdown and express tasks in natural language
* Not dependent on langauge-specific syntax

Even once you become a proficient programmer and feel comfortable writing code directly, more complex tasks can be facilited by pseudocoding the task. This also makes breaking code into discrete parts for collaborative coding much easier and can be used to talk about expected outputs and inputs for each subtask.

## 6.5 Good practices

While there is no set of standards for pseudocoding, there are ways to make it more effective. The most important thing you can do is to be consistent. When you pick a term (e.g. repeat or set), make sure that it has the same meaning each time you use it. It can be helpful to put these action words in all caps to highlight them. Some general rules for effective pseudocoding are:

* Make the first work the key action for the step and capitalize
* Only list one step per line
* Indent to show heirarchy of actions and improve readability
* End multiline blocks of code with END
* Make your pseudocode programming language independent
* Keep it simple, concise, and readable

## 6.6 Exercise

Let’s try a more complicated example to practice. Let’s say our task is to take a text file and return the line number of any line that contains a keyword. However, we want to ignore any comment line in the file, denoted by a “#” as the first character.

First, we’ll need a way to keep track of the line numbers

SET counter to 0

Next, we need step line by line through the file

FOR each line in the file

Then we need to check if the line is not a comment

IF the line does not begin with #

Next, we need to know if the keyword appears in the line

IF the line contains the keyword

Finally, if both conditions are true, we need to report the line number

PRINT counter

Now we can close the conditional statements

END IF  
 END IF

Now we need to advance our counter since we’re about to move to the next line

INCREMENT counter by 1

Finally let’s close the for loop

END FOR

Putting it all together, we have the following pseudocode

SET counter to 0  
FOR each line in the file  
 IF the line does not begin with #  
 IF the line contains the keyword  
 PRINT counter  
 END IF  
 END IF  
END FOR

And we’re done!

# 7 Fundamentals of Python - Intro & Data Types

## 7.1 Goal

Gain a basic familiarity with Python data types

## 7.2 Learning Objectives

After going through this module, students should be able to:

* Differentiate between the fundamental data types and structures in Python
* Define mutability
* Use variable assignment to store data/information within named variables

## 7.3 Resources

* [Object-Oriented Programming in Python is freely available documentation on programming in Python](https://python-textbok.readthedocs.io/en/1.0/index.html) and provides examples and thorough descriptions of fundamentals in Python programming. The table of contents is fairly well organized and has a lot of the buzz words contained in this module. There is more information covered in this book than will be required knowledge for you, but as a whole it likely will be a helpful resource to consult during the course.
* [Think Python 2nd Edition is freely available as web-based book](https://greenteapress.com/thinkpython2/html/index.html) and explains lots of Python vocabulary, syntax, and operations. The table of contents is quite thorough in describing each chapter’s contents. There is more information covered in this book than will be required knowledge for you, but as a whole it likely will be a helpful resource to consult during the course. This resource does not have an easy means to navigate back to the table of contents though.
* There is a [python syntax cheatsheet](http://bxlab.github.io/cmdb-bootcamp/cheatsheet/python.html) that may be useful, especially throughout the course.
* Other website resources will be linked throughout this module for specific topics. If a resource uses terminology or functionality that you are not expected to learn, we will note it after linking the resource.
* Use of all linked resources is optional but strongly recommended.

## 7.4 Syntax

Python makes use of indentation to define blocks of code. This means that you cannot mix spaces and tabs (4 spaces are recommended). This makes Python easy to read, but requires you to keep track of the indentation level when things become very nested.

## 7.5 Data types and structures

“Data” is information – information that you want to process or perform operations on, simplify, visualize, etc. This information must be stored within Python scripts or else read in from files (csv, txt, etc.). Python stores and represents the data as different data types or within different structures. Another word for data in python is “object”. You often want to explicitly make sure that the data is stored as a specific type so that you can perform specific operations on it (e.g., do specific things with or manipulate/analyze the data in a specific way). Math operations can be performed with integers or floats, but not strings. You can index with integers, but not floats, etc. Terms like these specific data types will be explained in more detail next. Terms like “index” will be explained in more detail in a few sections.

There are many data types and structures that are used in Python.

### 7.5.1 Integers:

* Integers are whole numbers
* These whole number are typed with just the numbers and do not include commas (,) or periods (.).
* Examples:
  + -2
  + 181
  + 10000056
  + 0

### 7.5.2 Floats:

* Floats are numbers that have fractional or decimal parts.
* These are typed with numbers and do have periods (.) or forward slashes (/) in them.
* Examples:
  + -2.5
  + 8/2
  + 79/3

### 7.5.3 Strings:

* Strings are words, phrases, or just single characters that are strung together and enclosed by single (') or double (") quotes.
* Examples:
  + “acgt”
  + “Banana”
  + “Hello, World!”
  + “1”
  + ‘The quick brown fox jumped over the lazy Gibbs’

### 7.5.4 Booleans:

* Booleans are variables that store True or False.
* Unlike integers, floats, and strings which can have many possible values, data that is a boolean data type is either True or False.

### 7.5.5 Lists:

* Lists are a data structure, specifically, collections of items that are enclosed in brackets. ( [ and ] ).
* In Python, lists can be homogeneous, and contain items that are all the same data type (e.g., a list of strings or a list of integers); or lists can be heterogeneous, containing items that are different data types.
* In Python, lists are also ordered, meaning that the elements or items contained in the list have a specific position associated with them.
* Examples:
  + [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
  + [“B”, “a”, “n”, “a”, “n”, “a”]
  + [“a”, 1, “c”, 0, “g”, 1, “t”, 1]
  + [“Hello,”, “World!”]
  + [“The”, “quick”, “brown”, “fox”, “jumped”, “over”, “the”, “lazy”, “Gibbs”]

### 7.5.6 Sets:

* Sets are another data structure, specifically, collections of unique (or non-duplicated) items that are enclosed in curly braces ( { and } ).
* To make a list into a set, the repeated items from the list are removed. For example, the list [“B”, “a”, “n”, “a”, “n”, “a”] would be the set {“B”, “a”, “n”}. One super cool thing is if you define your set to be {“B”, “a”, “n”, “a”}, Python will simplify it be {“B”, “n”, “a”}. (Also, note this is an example of how the set is unordered.)
* Sets are useful if you want to track all the unique elements present in a large dataset and don’t care if or how many times each element is repeated.
* Note that sets do not have any order so if you print a set out, the order of elements may change

### 7.5.7 Dictionaries:

* Dictionaries are another Python data structure, specifically collections of paired key,value items that are enclosed in curly braces. ( { and } ).
* By paired key,values we mean that a specific key is used to reference some value. For example, in the following dictionary dog\_favorite\_toys, "Gibbs" is a key used to refer to the value "football" because his favorite toy is a squeaking football.

dog\_favorite\_toys = {"Gibbs": "football",  
 "Duke": "squirrel",  
 "Barnaby": "frisbee"}

* What previously listed data type is each of the keys and values within that dictionary?
* Answer
* Strings
* Note that the key and value in a pair are separated by a colon (:) and the pairs are separated from each other by a comma (,)
* While sets are useful if you want to track all the unique elements present in a large dataset but don’t care if or how many times each element is repeated, dictionaries allow you to track the presence and associate further information like how many times the element is present. The key would be the element (like a gene ID), and the value would the number of times that element (gene ID) is present in the dataset. The key has to be unique within the dictionary. The values may not be unique, like multiple gene IDs may be present in the dataset the same number of times.
* Like sets, dictionaries do not have an order and are more like a jumbled bucket of information

### 7.5.8 Data represented or stored as an unintended data type:

Note: While we know that -2 is an integer, if you represent this integer in the following ways, it is not considered an integer by Python:

* -2.0
* -2/1
* “-2.0”

In the first case, Python considers these representations to be floats, which can still be used for mathematical operations, but couldn’t be used to index.

The second case is interpreted by Python as float division even in the case of two integers, one that divides evenly into the other. In order to do integer division, you need to use //. This tells Python to return an integer and drop any remainder.

5//2 = 2

In the third case, Python considers the data to be a string, and it cannot be used in mathematical operations unless it is converted to a numerical (integer or float) data type. Examples of converting with the int() and float() functions are available in the [Python syntax cheatsheet](http://bxlab.github.io/cmdb-bootcamp/cheatsheet/python.html) that was mentioned earlier. Often when reading data into Python from a file, numbers will be interpreted by python as strings and need to be converted to the correct data type.

### 7.5.9 Immutability

What does it mean for a data type to be immutable? It means that the value of an object with that data type cannot be changed without creating a new object.

### 7.5.10 Immutable data types

* Strings
* Integer
* Float
* Boolean

All other listed data types and structures from the previous section are mutable or can be changed. Practically speaking, Python is smart enough to recreate a variable but reuse the variable name when needed, such as

A = A + 1

## 7.6 Variables

What are variables? A Python variable is a symbolic name that is a reference to a data object. Once a data object is assigned to a variable, you can refer to the object by that name, retrieving it many times. While data objects can exist in Python without being stored in variables, you can’t reference and retrieve the data repeatedly without storing it in a variable. Within Python, you can have many variables each storing different values, and you distinguish between these variables according to their names.

The variable name is set within a variable assignment statement. We use an = (equals sign) to store data in a specific variable, associating that variable name with the value of the data. Then the variable can be referenced throughout a script. To store data in a variable, the variable name is on the left of the equals sign, and the data is on the right, like the following pattern:

variable\_name = "some data"

You’ve already seen an example of this with the dictionary dog\_favorite\_toys above. The data we are storing is saved in a variable named dog\_favorite\_toys. By using variables to store data, you can reference the variable later and use or modify the data stored in the variable. Assigning a value to a variable name is also called “defining” a variable or “initializing” a variable.

This variable assignment statement is considered an “expression” in Python. We’ll discuss more expressions in the next section of this module.

Variables can be overwritten, either intentionally or unintentionally. So for an example intentionally changing a variable’s value, consider that you may be looking through results of a genetic screen, wondering if a specific gene of interest was affected by the screen. You would store the gene of interest in a variable named gene\_of\_interest, and you set a boolean variable gene\_found to False because you don’t know if the gene of interest is in the large file. The script then looks through the file as long as the gene of interest isn’t found (and gene\_found is False). Once the script finds the gene of interest, you want to overwrite the gene\_found variable or change its value to True.

gene\_of\_interest = "DENN1B"  
gene\_found = False  
#Lots of other code that makes the script load and look through a large file  
#While looking through the file, the script finds the gene\_of\_interest  
#Since the gene was found, overwrite the gene\_found variable to be True and stop looking through the file  
gene\_found = True

For an example of unintentionally changing a variable’s value, you may reuse a variable name later in a script. Then that variable will no longer reference the value which was stored earlier.

What can you name a variable? Anything?

* Variable names should be meaningful and provide a description of what the variable contains. (However, comments within code can be used when defining a variable to accomplish this, especially if variable names are becoming unnecessarily long.)
* Note that while the name of the variable should be meaningful, the name doesn’t matter in how Python treats the variable. I could name a variable filename and store an integer in it and Python will treat the variable as the numerical value it has, not a string filename like its name suggests.
* Variable names can contain numbers (e.g., 0, 1, 2, …), but they cannot start with numbers. (i.e., file1 is an acceptable variable name, but 1file is not)
* There are “keywords” in Python 3 that are used in expressions and have very specific functionality associated with them. They cannot be used as variable names. You don’t need to memorize the list, but you [can find the list of keywords here](https://docs.python.org/3/reference/lexical_analysis.html#keywords)
* The case of the variable is important. myVariable is a different variable name than myvariable.
* It is acceptable to use a combination of upper and lowercase letters in variable names as well as underscores (\_). This can be helpful for longer variable names like gene\_of\_interest.

## 7.7 A deeper dive into lists

Let’s revisit lists. While we’ve seen how to create them, we also need to know how to alter them. Since a list is a mutable data type, we can alter it without being recreated.

In order to change a single element of a list, you simply need to reference the specific position using its “index”.

mylist = [1, 2, 3, 4]  
mylist[0] = "a"  
print(mylist)

## ['a', 2, 3, 4]

You’ll notice that the position numbering starts at zero instead of one. We’ll talk more about indexing later.

What about adding items to a list? Lists have several built-in methods (a function built-in to a class, and yes, a list is a class. It’s an object-oriented langauge thing). We will be using the append method.

mylist.append(5)  
print(mylist)

## ['a', 2, 3, 4, 5]

As you can see, we’ve added an element to the end of our list. Another way to add elements is to combine lists. To do this, we can used the plus sign.

print(mylist + [6, 7])

## ['a', 2, 3, 4, 5, 6, 7]

But what if we wanted to put an item in the middle of the list? We could use the list method insert. This places the item in the given position and shifts everything higher than that over.

mylist.insert(3, 'b')  
print(mylist)

## ['a', 2, 3, 'b', 4, 5]

We can also look for items in a list. If we wanted to search for “b”, we could use the index method.

print(mylist.index("b"))

## 3

This prints out 3, the index of “b” in the list. We can also remove items from a list either with the pop or remove methods. pop takes an index of the item to be removed while remove takes the object that you want to remove.

mylist.remove("b")  
print(mylist)

## ['a', 2, 3, 4, 5]

mylist.pop(0)

## 'a'

print(mylist)

## [2, 3, 4, 5]

The last method we’ll look at is sort. Like its name implies, this reorganizes the list in place. However, you cannot mix strings and numbers when sorting. So sorting [3, 2, 1] will work, but not [3, 2, "a"].

# 8 Fundamentals of Python - Expressions, Built-in Functions, and Conditionals

## 8.1 Goal

Understanding how to compare variables and selectively execute code

## 8.2 Learning objectives

After going through this module, students should be able to:

* Perform equality and inclusion testing between variables
* Demonstrate how to use common built-in functions, including printing and opening files
* Describe the use of conditionals to make choices within scripts

## 8.3 Operators and expressions

What is an operator? In python, an operator is some special symbol or keyword that designates that some specific computation should bbe performed. These include

* arithmetic operators for addition (+), subtraction (-), division (/), etc.
* comparison operators for testing equality of any value as well as whether numbers are greater than, less than, or equal to other numbers (== )
* logical operators for boolean values (and, not, or, etc.).

What is an expression? In python, an expression is some combination or sequence of values or variables and operators.

What is a built-in function? There are common built-in functions that are available to use in Python to perform specific tasks. These functions have names which are followed by parentheses. If the function takes any arguments that affect its behavior, these are included inside the parentheses when calling or using the function. Examples of built-in functions include print(), len(), etc.

## 8.4 Testing equality and inclusion

You can test whether the value of two objects are equal to one other by testing equality with a double equals sign (==) operator. Therefore an expression to test equality would look like the following pattern:

variable1 == variable2

Testing equality expressions are evaluated by Python, (i.e., Python finds the value of the expression) and return a boolean, so either True or False.

Here’s an example of a testing equality expression that will be evaluated as True.

3 == 3.0

## True

And here’s an example of a testing equality expression that will be evaluated as False.

dog = "Gibbs"  
dog == "dog"

## False

### 8.4.1 in

You can test whether some item is contained in a list or as a dictionary key, etc. using the in operator. Again, an expression that contains the in operator will be evaluated and return either True or False.

The structure for this expression follows this pattern:

variable in variable\_for\_data\_structure

testlist = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]  
7 in testlist

## False

testlist = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]  
8 in testlist

## True

dog\_favorite\_toys = {"Gibbs": "football",  
 "Duke": "squirrel",  
 "Barnaby": "frisbee"}  
"Gibbs" in dog\_favorite\_toys

## True

### 8.4.2 not or !

If you add not before in, you can test whether an item is not contained in some collection of items. While 7 in testlist was False, because 7 was not in the collection of items in testlist, 7 not in testlist should be True, because 7 is in fact not in the testlist collection.

testlist = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]  
7 not in testlist

## True

Or if you have a boolean variable variableName and you put not in front of the variable, then not variableName will take on the opposite value of variableName. Below a is a standin for the boolean variable.

| a | not a |
| --- | --- |
| T | F |
| F | T |

In a similar manner, while testing for equality or inequality of two variables or values, rather than using the double equals (==) to test for equality, you can use an exclamation point and a single equals sign (!=) to test if two variables or values are not equal.

And while the value of the variable dog didn’t equal the string "dog", and the expression dog == "dog" evaluated to False, dog != "dog" will evaluate to True.

dog != "dog"

## True

### 8.4.3 and

Sometimes you need to chain expressions together to see if multiple things are True. You can use the and operator in between the expressions. Below the letters a and b are stand-ins for the expressions.

| a | not a | b | a and b |
| --- | --- | --- | --- |
| T | F | T | T |
| T | F | F | F |
| F | T | T | F |
| F | T | F | F |

### 8.4.4 or

Alternatively, you may chain expressions together to see if at least one is True. You can use the or operator in between the expressions. Below the letters a and b are stand-ins for the expressions.

| a | not a | b | a and b | a or b |
| --- | --- | --- | --- | --- |
| T | F | T | T | T |
| T | F | F | F | T |
| F | T | T | F | T |
| F | T | F | F | F |

## 8.5 Common Built-in Functions

### 8.5.1 print()

The print() function is used to display formatted messages onto the screen or in a file. The function is also very useful in debugging to see what values certain variables are, their type, their length, etc when code isn’t behaving as expected.

print("Hello, World!")

## Hello, World!

the\_temperature\_today = 88  
print("The temperature outside in Baltimore today is", the\_temperature\_today, "degrees F.")

## The temperature outside in Baltimore today is 88 degrees F.

### 8.5.2 open()

In Python, you can open a file with data to read by passing the filename to the open() function. We will be using this function as we recreate the head bash tool later.

### 8.5.3 len()

Data types and structures that are collections of items (e.g., a string which is a collection of characters strung together; a list which is a collection of items, a set which is a collection of non-duplicated items, and a dictionary which is a collection of key-value pairs) have a length associated with them. This is the number of items in the collection. you can find the length of variables of these types using the Python function len().

len("acgt")

## 4

dog\_favorite\_toys = {"Gibbs": "football",  
 "Duke": "squirrel",  
 "Barnaby": "frisbee"}  
len(dog\_favorite\_toys)

## 3

len([0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144])

## 13

## 8.6 Conditionals

Conditional statements are used in python to check the value of a boolean variable or expression and specify what behavior or path of steps the script should follow depending on whether the boolean/expression was True or False. An example of an everyday conditional you may think through is like the following:

I am about to leave. Is it raining outside right now? Yes –> Grab an umbrella and leave

In Python, this could look something like the following if we have the boolean variable rainingNow.

rainingNow = True  
if rainingNow:  
 print('Grab the umbrella')

## Grab the umbrella

In the above code example, the boolean variable for whether it is currently raining is set to True, and the indented code under the if statement is executed. Also note that the conditional statement ends with “:”. This is required to denote the end of the conditional statement.

rainingNow = False  
if rainingNow:  
 print("Grab the umbrella")

In the same above code example, the only change is that the boolean variable for whether it is currently raining is set to False, and the indented code under the if statement is not executed/you don’t see the print statement.

In general, when using conditional statements, the boolean values are checked and anything indented under True boolean expressions/variables wll be executed linearly by the script. In python, you can have an if statement ( if booleanVariable:) without an else statement. Or you can have an if statement followed by an else statement. In that case, anything indented after the else statement will only be executed by the script if the value of the boolean variable is False.

rainingNow = False  
if rainingNow:  
 print("Grab the umbrella")  
else:  
 print("Leave without the umbrella")

## Leave without the umbrella

However, there may be times that there are other things to consider in addition to the initial if statement, things that you would want to think about when the initial if statement boolean variable or expression or condition is False. Let’s extend the raining example.

I am about to leave. Is it raining outside right now? Yes –> Grab an umbrella and leave No –> Will it be raining at some point while I am outside?     Yes –> Grab an umbrella and leave     No –> leave

In Python, this could look something like the following if we have the boolean variables rainingNow and rainingLater. We add an elif statement that will be checked if the first variable is False, but will be ignored if the first variable is True:

rainingNow = True  
rainingLater = True  
if rainingNow:  
 print("Grab the umbrella for now")  
elif rainingLater:  
 print("Grab the umbrella for later")  
else:  
 print("Leave without the umbrella")

## Grab the umbrella for now

This little code block asks if rainingNow is True. If it is, then the script will conclude that you should grab the umbrella, and it won’t check the elif statement to see if that variable is True. If it is False, the script ignores what is indented under the if rainingNow: and will check the value of rainingLater variable. If that’s True, the script will conclude you should grab the umbrella. But if that is also False, then, the script again ignores what is indented under the elif rainingLater: statement and the script goes to the else statement and concludes that you leave without the umbrella.

rainingNow = False  
rainingLater = True  
if rainingNow:  
 print("Grab the umbrella for now")  
elif rainingLater:  
 print("Grab the umbrella for later")  
else:  
 print("Leave without the umbrella")

## Grab the umbrella for later

Note here in this example how the print statements include the “for now” and “for later” phrases so we can distinguish which conditional statement was executed. Unique print statement usage is a tool you can use in debugging scripts.

Sections 5.2 - 5.7 on conditionals in the Think Python resource are especially helpful in looking at conditional statements.

# 9 Fundamentals of Python - Iteration, Indexing, and for Loops

## 9.1 Goal

Understanding how to access data in Python data structures through indexing and for loops

## 9.2 Learning objectives

After going through this module, students should be able to:

* Define iteration
* Define and use indexing
* List ‘iterable’ data types
* Use a for loop

## 9.3 Iterating

What is iterating and why do we want it? Iterating is going through a collection of items item-by-item. You likely want to use iterating if you want to apply the same set of operations on each item, or scan through the collection of items and catalog some information about the collection. We’ll see an example of use a for loop structure to iterate through lists in a few sections.

#### 9.3.0.1 Iterable data types

Iterable data types are those which are a collection of items that can be iterated through item-by-item:

* Strings
* List
* Sets
* Dictionaries

## 9.4 Indexing

What is indexing? Indexing is a way to refer to individual elements in an ordered, iterable object (like lists or strings) by their position or location within the ordered, iterable object. An index or these positions/indices are specifically integer data types. Floats, strings, and other such data types cannot be used to index.

The indexing “operator” in python is brackets, []. So like the arithmetic, comparison, and logic operators discussed earlier, the indexing operator is a special symbol(s) with a pre-determined set of behaviors associated with it. The brackets directly follow the name of whatever ordered, iterable object you wish to index, and the index position(s) you wish to extract or change are included in between the brackets. If you try to use an indexing operator to index an unordered object like a set, you will see the following error: TypeError: 'set' object is not subscriptable.

In Python, indexing starts from 0. So for a list of days of the week dow = ["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"], even though Sunday is the first day of the week, and the first element in the list, its position or index is 0. And Saturday, the last or 7th day of the week has an index or position of 6. If you try to index this list with a position of 1 (e.g., dow[1]), that position refers to Monday. And if you try to index this list with a position of 7 (e.g., dow[7]), you’ll see the following error that says there is no index of 7 in the data structure or list dow: IndexError: list index out of range. If you want to extract an element from a list and store it in a variable, then you’d use the following pattern like we saw earlier with variable assignment: last\_day\_of\_week = dow[-1]. Note that the list name, indexing operator, and index position are on the right side of the equals sign. Note that the dow list is unchanged compared to its original identity.

Recall that lists are mutable which means that in addition to using indexing to extract information from the list, grabbing a list element that is in a specific position, you can also use indexing to change the list and overwrite information within the list, modifying a list element that is in a specific position. Becasue of this, if you want to modify a specific list position, where would the list name, indexing operator, and index position be in relation to the equals sign in a variable assignment expression? For example, let’s say that you decide to outlaw Mondays and replace them with a second Thursday, the data that you want to store in the variable is "Thursday" and the variable that you want to store it in is dow but specifically dow at position 1, so dow[1]. Then your variable assignment statement would be dow[1] = "Thursday" and the dow list is changed compared to its original identity: ['Sunday', 'Thursday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

### 9.4.1 Slicing

What is slicing? Slicing is a fancy way of indexing to extract a subset of elements from an interable object using a range of the positions within the ordered,iterable object.

The starting index is inclusive and the ending index is exclusive. That means that every position corresponding to a number in between the starting index and the ending index will be included and the starting index position is included, but the ending index position is not included. This also means that you can tell how many positions you are requesting by subtracting the starting index from the ending index.

Here’s a really cool [suggested resource about slicing](https://towardsdatascience.com/the-basics-of-indexing-and-slicing-python-lists-2d12c90a94cf) with examples and visuals.

* Terms within this resource that you are not expected to understand include “list comprehension” and “numpy Arrays” and “pandas DataFrames”.
* Another term within this resource, for loops, will be explained in the next section of this module.

## 9.5 for loops

for loops are a very commonly used tool for iterating in Python and are often necessary when processing file inputs.

A for loop has several components.

1. The for statement
2. The body of the for loop
3. Any variable initialization before the for statement

The first two of these three components are part of every for loop. The last component is only sometimes required. We’ll begin by discussing the for statement and the body of the for loop and then will provide examples of loops that have variable initialization before the statement and those that do not.

### 9.5.1 The for statement

The for statement has 5 parts to it, in order:

1. the word for
2. the variable name for whatever is being extracted within each iteration or step of the for loop
3. the word in
4. the iterable variable name for the variable from which something is being extracted within each iteration or step of the for loop
5. A colon, :

There 3 common for statement patterns in python. Try to connect the 5 parts of the for statement to these patterns.

1. for value\_from\_iterable\_variable in some\_iterable\_variable:

This statement uses the above pattern (its setup broken down below) in order to work with the items contained in a collection of items item-by-item.

* the word for
* then names the variable that each item from the some\_iterable\_variable will be stored in: value\_from\_iterable\_variable
* the in membership operator, not to check containment like the expressions we saw previously, but to specify that we want items contained in the some\_iterable\_variable
* the name of the variable for the collection over which we want to iterate and extract its items item-by-item: some\_iterable\_variable
* A colon, :

1. for index in range(len(some\_iterable\_variable\_name)): or for index in range(some\_integer):

The range() function is a built-in Python function that is used to return a list of numbers from 0 up to, but not including some integer number. You would use this above statement pattern (its setup broken down below) in order to work with the ordered position numbers rather than the items themselves that are stored in a collection of items. With indexing, you could still access the items item-by-item, but the variable name for whatever is being extracted is storing integers, the index positions, not the items.

* the word for
* then names the variable that each item from the range() function will be stored in: index
* the in membership operator to specify that we want items contained in the list the range() function returns
* the range() function that returns a list of numbers either specified by the len() function and the length of some\_iterable\_variable\_name or just some integer value (some\_integer)
* A colon, :

1. for index, value\_from\_iterable\_variable in enumerate(some\_iterable\_variable\_name):

This third statement is a combination of the first two statements where for each iteration or step in the for loop, you have access to both the value and the index of the value from the iterable variable at the same time. This is achieved because the pattern uses a special base python function enumerate(). You can [read more about this function from the online python manual](https://docs.python.org/3/library/functions.html#enumerate), but suffice it to say that the function takes in an iterable object like a list or a string. Then rather than just returning the elements of the iterable object, it also returns the corresponding indices enumerated or counted out for each of the elements in the iterable object.

For example, if we have a list test\_list = ["daisy", "poppy", "daylily", "sunflower", "begonia"]. In a for loop statement following the first pattern, for flower in test\_list:, only elements from the list are returned, specifically to the variable flower. And the flower variable will be “daisy” in the first iteration of the for loop, “poppy” in the second iteration, “daylily” in the third iteration, “sunflower” in the fourth, and finally “begonia” in the fifth. If we instead follow the third pattern and use the enumerate function, specifically, for index, flower in enumerate(test\_list), the flower variable will take on the same values for each iteration of the loop as the first pattern example. The index variable will take on the values 0, 1, 2, 3, and 4 respectively.

The setup is broken down below for this pattern

* the word for
* the names the variables that each index and each item from the some\_iterable\_variable\_name will be stored in: index, value\_from\_iterable\_variable
* the in membership operator to specify that we want items contained in the output from enumerate()
* the enumerate() function with the some\_iterable\_variable\_name over which we want to extract items and their indexes item-by-item.
* A colon, :

### 9.5.2 for loop body

**Indentation**

The for loop body is going to be indented below the for loop statement. This is necessary because indenting tells Python what is done as part of the for loop, and when to re-start the next iteration of the for loop.

As an example, consider this for loop:

dogowners = ["Kate", "Dylan" ,"Mike", "Andrew"]  
for index, dogname in enumerate(["Gibbs", "Chela", "Barnaby", "Duke"]):  
 print("The owner of", dogname, "is", dogowners[index])

## The owner of Gibbs is Kate  
## The owner of Chela is Dylan  
## The owner of Barnaby is Mike  
## The owner of Duke is Andrew

print("What other information should we share about these dogs? Size? Breed? Favorite toy?")

## What other information should we share about these dogs? Size? Breed? Favorite toy?

First, notice that the print statement that is indented under the for statement will be printed four time, each one with a different dog name, and a different dog owner’s name. But the print statement that is not indented under the for statement, and instead is aligned with it, will only be executed once, after the other 4 lines have printed and the for loop is complete.

Second, let’s go back a step and break down the pattern of the for statement.

* Which pattern of the 3 was used?
* Why do you think that pattern was used?
* Is there another pattern that could have been used as effectively?

Answers:

* The third pattern
* We wanted access to each list item and the position of each list item so that we could index a second matching list
* The second pattern could be used just as effectively, and we would index both lists instead of just one.

Like with conditional statements, the order of execution within the body of the loop (all lines that are indented) follows a downwards linear pattern. Lines listed earlier/first will be executed first and lines listed later will be executed later. Once every iteration of the for loop is complete, then lines outside of the for loop will be executed, like how the final print statement that asks what other dog innformation should be shared is only displayed after the for loop body is complete.

### 9.5.3 initializing variables before the for loop

An example of when you might want to initialzie variables before the for loop that will be edited within the for loop body is when you may want to count how many times a condition is met within a for loop. In such a case, you would initialize a variable before the for loop and then add one to the counter within the for loop body every time the condition is met.

This condition could be as simple as number of times the for loop iterates and so you add one to the counter every single iteration:

dow = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']  
counter = 0  
for day\_of\_week in dow:  
 counter = counter + 1  
print(counter)

## 7

Or it may be more complex, like seeing if the day of the week starts with the letter T or S. Here we’ll use the first pattern for the for statement because we’ll want to have access to the actual day of the week name, and then within the for loop body, we’ll use an if statement with two expressions returning True or False and an or logical operator between the two expressions. To see if the day of the week starts with the letter T or S, we’ll use a python built-in function startswith(). Note that we’re re-initializing the counter variable to 0 outside of the for loop and adding one every time the a or b expression evaluates to True.

counter = 0  
dow = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']  
for day\_of\_week in dow:  
 if day\_of\_week.startswith("T") or day\_of\_week.startswith("S"):  
 counter = counter + 1  
print(counter)

## 4

### 9.5.4 Pseudo-coding a for loop

When writing pseudo-code and knowing that you’ll need to use a for loop, we find that it’s easiest to go back to the basic structure of a for loop.

1. Start by filling in your for statement

* Write the for, in, and : and leave blanks for what you’re extracting information from (part 5) and what you’re extracting (part 3)
* Then start on part 5.
  + Add in the variable name for what you’re extracting information from
  + Do you want to index or just extract a value or do both?
* Then fill in informative variable names for part 3 of the for statement

1. Outline in words what you want to do within the for loop body

* remember to indent what should be done for each iteration of the loop
* use your for statement part 3 variable names as much as possible within this outline
* remember that order matters and the steps are going to be executed sequentially within the loop.

# 10 Fundamentals of Python - Importing and Commenting

## 10.1 Goal

Understanding the importance of code commenting and how to use external code

## 10.2 Learning objectives

After going through this module, students should be able to:

* Clearly annotate their code
* Load external code into their scripts

## 10.3 Commenting

While it may seem like extra work, adding context to code is **crucial**. It’s helpful to remind you as a programmer when you look back at an old script what you intended as well as making it much easier for someone else reading the code to follow your logic.

### 10.3.1 Adding comments to code

We can add comments to our code to explain what we are doing or what we want to do using a pound sign or the hash (#). You can either start a line with this or put it at the end of the line. Everything on the line after the pound sign (#) will be ignored by python and not executed, but allows readers of code to better understand what is happening.

## 10.4 Importing packages (AKA libraries or modules)

Finally, sometimes in Python we’ll use the import keyword to import packages that have pre-built functions that aren’t built-in Python functions, but nonetheless allow us to do specific things. A very common package that we’ll be using in this prep work is sys which allows us to take in commandline arguments when calling a bash script on Python. For plotting we would use a package called matplotlib.pyplot.

### 10.4.1 Importing specific functions and variables

Often you will not need to import an entire library or module, but rather a specific function or variable. For example, if we wanted to have access to the log function which is in the math library, we could either

import math  
a = math.log(5)

But we could also just get the log function using the from keyword.

from math import log  
a = log(5)

This is much easier to read and write. However this should be used with caution as some functions in different libraries have the same name and importing one will replace another function of the same name. But if we import and reference the module (i.e. using math.log), that problem is alleviated.

We can also import multiple functions/variables in a single line.

from math import log, exp, sin

### 10.4.2 Aliasing packages or modules

Sometimes it is too bulky to type out the complete name of the module and function that you have imported each time you wish to use it. Instead, Python has the as keyword which allows us to given modules or even individual functions alternate names. One that we will use repeatedly in this class is

import matplotlib.pyplot as plt

Anytime we type plt, Python interprets this as matplotlib.pyplot. So if we want to call the subplots function from this module, we can simply type plt.subplots().

Note that you should use this aliasing with caution. As with importing functions without reference to their module, aliases will replace another function or variable of the same name in your script.

# 11 Applied Python Exercise Outline

## 11.1 Goal

Outline recreating and extending the bash tool head in Python

## 11.2 Learning Objectives

After going through this module, students should be able to:

* Explain what the tool head does
* List the different tasks that make up head’s main function
* State what the assignment is following these modules

## 11.3 Intro

One of the strengths of Python that was previously mentioned is its versatility and how it can be used to extend and visualize the results of analyses performed with the command line. Therefore, the overarching goal of these prepwork modules is to recreate and extend the functionality of a common bash tool using Python. These modules will guide you through recreating head, adding some functionality outside of the tool’s basic behavior.

As a reminder, head is used to display the first n number of lines in a file. If no number is specified, the tools defaults to displaying 10 lines.

## 11.4 Coding Blueprint

In these modules, you will be guided through writing code that does each of the following to recreate head:

1. displaying every line in an input file
2. displaying just the first line in an input file
3. displaying a specified number of lines from the beginning of an input file
4. displaying a specific number of lines from the beginning of an input file, specifying the number outside of the code itself as an additional input
5. displaying a default number of lines from the beginning of an input file if another number isn’t specified as an additional input

Then, you will be guided through writing code that extends the recreated head program such that additionally it can

1. skip a file header before displaying the output

## 11.5 Final Assignment

Finally, you will be asked to take your recreated head program and edit it such that it would tail an input file instead.

Each of these 7 tasks will focus on using a specific python fundamental or data type like for loops, conditionals, or lists. Each guided section will help you to break the task into small manageable pieces, write pseudocode for these smaller tasks, and then build the code. For the final task, you will be asked to both pseudocode and code on your own.

# 12 Applied Python Exercise 1

## 12.1 Goal – Print every line in a file

Write Python code that works towards recreating the Bash tool head, displaying every line in a file: random\_snippet.vcf

## 12.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + variable assignment
  + strings
  + opening a file
  + for loop
  + printing output

## 12.3 Coding Blueprint

Within this module, you will write code that displays every line in an input file. To complete this goal, consider the most basic tasks that would be required.

First, we need to SET the input file Second, FOR every line in the open file   PRINT the line END FOR

### 12.3.1 SET the input file

Let’s focus on that first step: SET the input file. One important thing when breaking down tasks into smaller, manageable steps is asking yourself what information is available in your script, and what information you need to define or store in variables. The filename is something that will need to be assigned to a variable so that it is available information.

### 12.3.2 FOR every line in the open file …

Let’s focus on the FOR portion of the pseudocode now, specifically building the for loop structure by considering each of the 3 components discussed earlier:

1. The for statement

Which for statement pattern best fits the task you want to accomplish of printing every line in the file? We’ll see as we build the code that the first pattern, for line in file, will work very well for this task because all we need is to work with the items (lines) contained in a collection (the file) item-by-item.

However, we will have to slightly edit this pattern to make sure we’re working with the contents of the file, and not the contents of the string variable storing the filename. If we don’t open the file and the file variable refers to the string of the filename, then line will just be the letters of the filename one at a time. Therefore, we’ll want to make sure that file refers to the open file.

### 12.3.3 PRINT the line

1. The body of the for loop

We’ve already defined this well - we want to PRINT the line.

1. Any variable initialization before the for statement

Is there any need to initialize a variable before the for loop that you will edit within the body of the loop? In this case, because we’re printing every line in the file, there’s no need to initilalize a variable beforehand.

## 12.4 Building the code

**Create a new Python script**

### 12.4.1 SET the input file

To SET what the input file is, we will create a variable which stores the string of the filename for the input file.

Consult the notes on variable assignment earlier to SET a variable equal to the input file. Within your Python script, write a line of code that does this.

ANSWER:

filename = "random\_snippet.vcf"

### 12.4.2 FOR every line in the open file

Then, that variable is just storing the name of the file. The actual file contents, or the lines of the file, aren’t available to Python until the file is opened. Consult the notes on the built-in Python function open() and use the determined structure of the for statement from the pseudocode section above to open the file and begin the for loop architecture. Within your Python script, write a line of code that does this.

ANSWER:

for line in open(filename):

### 12.4.3 PRINT the line

Now that we have the first component of the for loop, we can write the body of the for loop so that each line is printed as output. Consult the notes on the print() function indenting the body of the for loop under the for statement. Within your Python script, write a line of code that does this.

ANSWER:

print(line)

If we were just to use the code above, there would be a blank line printed at the end of each line from the file. We can remove the blank lines from the output by using another built-in Python function specifically strip(). Newline characters which apepar as blank lines in the output are either "\n" or "\r" within bioinformatic files. Since we’re not certain which of these our input file has, we’ll strip both of these characters off the end of each line using strip("\r\n"), specifically telling the strip function to remove both of those characters. We can tell the strip function to remove those characters from each line using the following syntax: line.strip("\r\n"). Use of this function with this syntax will remove or suppress those blank lines in the output. Edit the line of code within your Python script to do this.

ANSWER:

print(line.strip('\r\n'))

Finally, within your Python script, you should have the three lines of code together which makes our complete intended goal code – code that prints every line in a file.

**Use the command line to run the Python script and look at the output.**

## 12.5 Complete Intended Goal Code

ANSWER:

filename = "random\_snippet.vcf" #SET the input filename  
for line in open(filename): #FOR every line in the open file  
 print(line.strip('\r\n')) #PRINT the line

# 13 Applied Python Exercise 2

## 13.1 Goal – Print the first line in a file

Write Python code that works towards recreating the Bash tool head, displaying only the first line in a file: random\_snippet.vcf

## 13.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + variable assignment
  + strings
  + opening a file
  + for loop
  + integers
  + logical expression
  + conditional statement
  + printing output

## 13.3 Coding Blueprint

Let’s start with and edit the pseudocode from the last module to meet the needs of this module.

Last module’s pseudocode:

First, we need to SET the input file Second, FOR every line in the open file    PRINT the line END FOR

In this module, we want to only print the line if it’s the first line of the file. Keeping the for loop, edit the pseudocode, inserting a step that asks which line position it is numerically in the file.

ANSWER:

First, we need to SET the input file Second, FOR every line in the open file    IF the first line      PRINT the line    END IF END FOR

### 13.3.1 SET the input file

Much like how we reused the pseudocode of the last module, we’ll be able to reuse much of the code from the last module. Do you think that defining a variable with the input file name is one of those reusable steps?

ANSWER:

Yes, it is

### 13.3.2 FOR every line in the open file

Let’s again focus on building the for loop structure by considering the 2 components of the loop used in the first module

1. The for statement

Do you think that the for statement from the last module is the best pattern for this problem? Consider that the for statement in the last module only provided you access to each line (or item) in the file (or collection of items), but did not provide you any information on which line position numerically it was in the file.

ANSWER:

We should use the third for statement pattern instead which provides both the item and the item’s position or index.

### 13.3.3 IF the first line and PRINT the line

1. The body of the for loop

Considering that the body of the for loop is where the pseudocode was added, we also need to add to the body of the for loop. However, does the code that we already have from the first module still work well for this module?

ANSWER:

Yes, it does.

What can we use to see if the index or position of the line is the first line in the file?

ANSWER:

We can use a conditional and logical expression such that we’ll PRINT the line, but only if it’s the first line in the file.

## 13.4 Building the code

**Create a new Python script**

### 13.4.1 SET the input file

To SET what the input file is, we will use the variable assignment code from the first module. Within your Python script, write that variable assignment expression.

ANSWER:

filename = "random\_snippet.vcf"

### 13.4.2 FOR every line in the open file

Consult the notes on the third pattern of the for statement and edit the first modules for statement code to meet the needs of this module. Within your python script, write the revised for statement.

ANSWER:

for i, line in enumerate(open(filename)):

### 13.4.3 IF the first line

Next, edit the body of the for loop from the first module such that it asks if the line is the first line. Consult your notes on indexing to see what integer you want the index to be equal to (e.g., 0 or 1). Write the conditional and logical expression code within your Python script within the for loop body.

ANSWER:

if i == 0:

### 13.4.4 PRINT the line

Finally, reuse the code from the first module to print the line (when it is the first line). Add this print() statement to your Python script, indenting correctly under the conditional statement.

ANSWER:

print(line.strip('\r\n'))

Within your Python script, you should have the four lines of code together which makes our complete indended goal code – code that prints just the first line in a file.

**Use the command line to run the Python script and look at the output.**

## 13.5 Complete Intended Goal Code

ANSWER:

filename = "random\_snippet.vcf" #SET the input filename  
for i, line in enumerate(open(filename)): #FOR every line in the open file  
 if i == 0: #IF the first line  
 print(line.strip('\r\n')) #PRINT the line

# 14 Applied Python Exercise 3

## 14.1 Goal – Print some specified number of lines from the beginning of an input file: random\_snippet.vcf

Write Python code that works towards recreating the Bash tool head, displaying a specified number of lines from the beginning of a file.

## 14.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + variable assignment
  + strings
  + opening a file
  + for loop
  + integers
  + logical expression
  + conditional statement
  + printing output

## 14.3 Coding Blueprint

Let’s again start with and edit the pseudocode from the last module to meet the needs of this module.

Last module’s pseudocode:

First, we need to SET the input file Second, FOR every line in the open file    IF the first line      PRINT the line    END IF END FOR

In this module, we no longer want to print the line if and only if it’s the first line of the file. We now want to print the line if it’s any of the first several lines of the file, where the actual number of lines is specified or set or defined, much like how we define the input file name. Edit the pseudocode, adding one line, and editing another line to reflect this change in the goal.

ANSWER:

First, we need to SET the input file Next, we need to SET the desired number of displayed lines Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

## 14.4 Building the code

**Create a new Python script**

Much like how we added one line and edited one line of the previous module’s pseudocode, for the actual code, you will only need to add one line of code, and edit one line of code.

### 14.4.1 SET the input file

To SET what the input file is, we will use the variable assignment code from the first and second Python exercise modules. Within your Python script, write that variable assignment expression.

ANSWER:

filename = "random\_snippet.vcf"

### 14.4.2 SET the desired number of lines

To SET the desired number of beginning lines to display, write a variable assignment expression, using some integer value as the desired number. Add this new line of code within your Python script.

ANSWER:

n\_lines = 10

### 14.4.3 FOR every line in the open file

To loop through every line in the open file, use the for statement structure from the second Python exercise module. Write this line of code within your Python script.

ANSWER:

for i, line in enumerate(open(filename)):

### 14.4.4 IF a desired line (by its numerical position)

Next, edit the body of the for loop, specifically the conditional expression, such that it asks if the line is one of the desired beginning lines. Consult your notes on indexing in python to see what numerical comparison operator (e.g., <=, <, ==) is necessary. Add this edited line of code within the for loop body within your Python script.

ANSWER:

if i < n\_lines:

### 14.4.5 PRINT the line

Finally, resue the code from the first two Python exercise modules to print the line, adding this print() statement indented correctly under the conditional within your Python script.

ANSWER:

print(line.strip('\r\n'))

Within your Python script, you should have the five lines of code together which makes our complete intended goal code – code that prints a specified number of lines from the beginning of a file.

**Use the command line to run the Python script and look at the output**

## 14.5 Complete Intended Goal Code

ANSWER:

filename = "random\_snippet.vcf" #SET the input filename  
n\_lines = 10 #SET the desired number of lines  
for i, line in enumerate(open(filename)): #FOR every line in the open file  
 if i < n\_lines: #IF a desired line by its numerical position  
 print(line.strip('\r\n')) #PRINT the line

# 15 Applied Python Exercise 4

## 15.1 Goal – Print a specific number of lines from the beginnning of an input file, allowing the specification of the number of lines outside of the code by the user

Write Python code that works towards recreating the Bash tool head, displaying a user specified number of lines from the beginning of a user specified input file.

## 15.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + importing modules
  + variable assignment
  + list indexing
  + integer conversion
  + opening a file
  + for loop
  + logical expression
  + conditional statement
  + printing output

## 15.3 Coding Blueprint

Let’s again start with and edit the pseudocode from the last module to meet the needs of this module.

Last module’s pseudocode:

First, we need to SET the input file Next, we need to SET the desired number of displayed lines Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

Do we need to change this pseudocode at all or does it accurately reflect what we want to accomplish with our new script?

ANSWER:

No we don’t need to change the pseudocode from the last module because it still accurately reflects the goal we want to accomplish within this module.

## 15.4 Building the code

**Create a new Python script**

We may not need to edit the pseudocode for this module, but we will need to edit the code. The end users of tools don’t have access to the values of variables within code unless the variables can be passed information through the command line. So rather than defining the variables for the input filename and desired number of lines within the script as hardcoded, inflexible variables with specific values, we’re going to define the variables based on input arguments used when running the script, increasing the flexibility of the tool.

The sys module is useful for this in Python. Specifically, it allows you to use a structure called sys.argv which is a list that stores the command line arguments passed to a Python script. sys.argv[0] is always the script name with the additional information/arguments starting at index 1 or sys.argv[1].

When you use sys.argv in a script, it’s a good practice to include a USAGE statement comment at the beginning of the script showing how the script should be called. In our case, we want the user to pass the input filename as the first argument and the desired number of lines as the second: #USAGE: python scriptname.py input\_filename number\_lines\_to\_display (Note that calling the python interpreter is not included in the list since it’s before the script name).

So in building our code, we want to make sure to import the sys module and set a USAGE comment statement. Within your new Python script, do these two things.

ANSWER:

#USAGE: python scriptname.py input\_filename number\_lines\_to\_display  
import sys

### 15.4.1 SET the input filename

In the code for the past modules, we’ve set the filename as a hardcoded string variable. Now you want to set it as the first command line argument passed to the program when calling the script. Do this within your Python script.

ANSWER:

filename = sys.argv[1]

### 15.4.2 SET the desired number of lines

In the code for the last module, we set the desired number of lines as a hardcoded integer variable. Now you want to set it as second command line argument passed to the program when calling the script. Do this within your Python script.

ANSWER:

n\_lines = sys.argv[2]

However, we need the variable to be an integer, and the values in the sys.argv list are always stored as strings, even if they are numbers. Therefore, edit the line within your Python script to convert the value that is being stored to an integer.

n\_lines = int(sys.argv[2])

Now that we’ve managed to set the input file and the desired number of lines to values that are specified by the user on the command line, we can reuse the rest of the code for the next three steps that we used in the previous module.

### 15.4.3 FOR every line in the open file

To loop through every line in the open file, use the for statement structure from the second and third Python exercise modules. Write this line of code within your Python script.

ANSWER:

for i, line in enumerate(open(filename)):

### 15.4.4 IF a desired line (by its numerical position)

To ask if the line is one of the desired beginning lines, use the conditional statement structure from the third Python exercise module. Write this line of code within your Python script, indenting correctly under the for loop statement.

ANSWER:

if i < n\_lines:

### 15.4.5 PRINT the line

Finally, reuse the code from the first three Python exercise modules to print the line, adding this print() statement indented correctly under the conditional within your Python script.

ANSWER:

print(line.strip('\r\n'))

Within your Python script, you should have the seven lines of code together which makes our complete intended goal code – code that prints a user specified number of lines from the beginning of a user specified input file.

**Use the command line to run the Python script and look at the output**

## 15.5 Complete Intended Goal Code

ANSWER:

#USAGE: python scriptname.py input\_filename number\_lines\_to\_display  
import sys #import module  
filename = sys.argv[1] # SET the input filename  
n\_lines = int(sys.argv[2]) # SET the desired number of lines  
for i, line in enumerate(open(filename)): #FOR every line in the open file  
 if i < n\_lines: #IF a desired line by its numerical position  
 print(line.strip('\r\n')) #PRINT the line

# 16 Applied Python Exercise 5

## 16.1 Goal – print a default number of lines from the beginning of an input file if another number isn’t specified as an additional input

Write Python code that recreates the Bash tool head, displaying a default number of lines in an input file if a specific number of lines isn’t specified.

## 16.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + importing modules
  + variable assignment
  + list indexing
  + list length
  + integer conversion
  + opening a file
  + for loop
  + logical expression
  + conditional statement
  + printing output

## 16.3 Coding Blueprint

Let’s again start with and edit the pseudocode from the last module to meet the needs of this module.

Last module’s pseudocode:

First, we need to SET the input file Next, we need to SET the desired number of displayed lines Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

We do want to edit this past pseudocode, adding some clarity to how we’re going to SET the desired number of displayed lines. Specifically, we’re either setting the desired number as a user-specified input, or as a default number if no user-input is given. Edit that line of the pseudocode, expanding it with conditional expression language to reflect this change in the program behavior.

ANSWER:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

## 16.4 Building the code

**Create a new Python script**

### 16.4.1 Usage statement and module importing

We again want to import the sys module so that users specify the input file and can specify the desired number of lines (if they want). Within your new Python script, import the module.

We’ll want to update our usage statement comment to reflect that only the filename is required. Adding brackets ([, ]) around an argument description in the usage statement is the preferred way to say that the argument is not required as input. Within your Python script, update your previous usage statement from the fourth Python exercise module to show that the number of lines to display is no longer a required argument.

ANSWER:

#USAGE: python scriptname.py input\_filename [number\_lines\_to\_display]  
import sys

### 16.4.2 SET the input filename

Within your Python script, set the filename variable to be equal to the first command line argument like you did in fourth Python exercise module.

ANSWER:

filename = sys.argv[1]

### 16.4.3 IF the user-specified a desired number of lines to display

In the previous Python exercise module, we wanted the second command line argument to be the number of lines that would be displayed by the program. In this module, we want to see if the user provided a desired number of lines as a command line argument. To do this, let’s consider what information we know and what variables/objects/functions are available to us.

Command-line arguments are stored in the sys.argv object which we already said is a list. We’ve discussed previously that lists have a length associated with them where their length is equal to the number of elements in the list. Given that sys.argv[0] is always the name of the script, if no arguments are passed to a script when calling it from the command line, the length of the list would be 1. If only one argument is passed, then the length would be 2. And finally, if two arguments are passed, then the length would be 3. Consult your notes on the len() Python built-in function, and write code in your Python script that checks if the length of the sys.argv list suggests that the user specified a number of lines to display.

ANSWER:

if len(sys.argv) > 2:

### 16.4.4 THEN SET the desired number of lines

If the user did specify the number of desired lines, we want define the variable for the desired number of lines to equal this value. Within your Python script and indented under the conditional you’ve just written, follow the method we used in the fourth Python exercise module to define this variable.

ANSWER:

n\_lines = int(sys.argv[2])

### 16.4.5 OTHERWISE

If the user didn’t provide a specified number of lines, then we want alternate behavior in how we define the desired number of lines variable. Consult the notes on conditionals to see how to specify what structure should be used to specify behavior for when the if conditional evaluates to False. Within your Python script, aligned with the if, add the appropriate conditional statement.

ANSWER:

else:

### 16.4.6 THEN SET the desired number of lines to a default

In the third Python exercise module, we previously hardcoded the number of lines we wanted to display. Within your Python script, indented under the else statement, write code that defines the default number of lines as 10.

ANSWER:

n\_lines = 10

Now that we’ve managed to set the input file and the desired number of lines to values that are specified by the user or a default value, we can reuse the rest of the code for the next three steps that we used in the previous two Python exercise modules.

### 16.4.7 FOR every line in the open file

To loop through every line in the open file, use the for statement structure from the second, third, and fourth Python exercise modules. Write this line of code within your Python script.

ANSWER:

for i, line in enumerate(open(filename)):

### 16.4.8 IF a desired line (by its numerial position)

To ask if the line is one of the desired beginning lines, use the conditional statement structure from the third and fourth Python exercise modules. Write this line of code within your Python script, indenting correctly under the for loop statement.

ANSWER:

if i < n\_lines:

### 16.4.9 PRINT the line

Finally, reuse the code from the first four Python exercise modules to print the line, adding this print() statement indented correctly under the conditional within your Python script.

ANSWER:

print(line.strip('\r\n'))

Within your Python script, you should have the ten lines of code together which makes our complete intended goal code – code that recreates the Bash tool head – a program that prints a user specified number of lines from the beginning a user-specified input file, unless no specific number of lines is specified, then it prints a default of 10 lines.

**Use the command line to run the Python script and look at the output**

## 16.5 Complete Intended Goal Code

ANSWER:

#USAGE: python scriptname.py input\_filename [number\_lines\_to\_display]  
import sys #import module  
filename = sys.argv[1] #SET input filename  
if len(sys.argv) > 2: #IF user-specified number of lines provided  
 n\_lines = int(sys.argv[2]) #SET the desired number of lines  
else: #OTHERWISE  
 n\_lines = 10 #SET the desired number of lines to a default  
for i, line in enumerate(open(filename)): #FOR every line in the open file  
 if i < n\_lines: #IF a desired line by its numerical position  
 print(line.strip('\r\n')) #PRINT the line

# 17 Applied Python Exercise 6

## 17.1 Goal – recreate the Bash tool head in Python; adding some new functionality that skips header lines

Write Python code that recreates the Bash tool head, displaying a default number of lines in an input file if a specific number of lines isn’t specified; but extend the tool so that it skips header lines (specifically, for this example, lines that start with a pound sign (#)).

## 17.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + importing modules
  + variable assignment
  + list indexing
  + list length
  + integer conversion
  + opening a file
  + initializing a counter variable before a for loop
  + updating a counter variablbe within a the body of a for loop
  + for loop
  + chaining logical expressions
  + looking at the character a line starts with
  + conditional statement
  + printing output

## 17.3 Coding Blueprint

Let’s again start with and edit the pseudocode from the last module to meet the needs of this module.

Last module’s pseudocode:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

We do want to edit this past pseudocode, adding a condition that should be true before we print a line. Edit that line of the pseudocode, expanding it with logical expression language to reflect this change in the program behavior.

ANSWER:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE Then, FOR every line in the open file    IF a desired line (by its numerical position) AND NOT a header line (line doesn’t start with a pound sign #)      PRINT the line    END IF END FOR

We also can no longer assume that the numerical position of the line within the file corresponds to the number of lines that have been printed since we’re filtering out the header. Therefore, we need to track the number of lines that have been printed. Add two lines to the pseudocode that will help you do this, and further edit the conditional that we check before printing a line. As a hint for those, this is the first time we’ll be using the third optional component of a for loop: initializing a variable before the for loop (and then subsequently) updating it within the body of the loop.

ANSWER:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE SET counter of number of lines that have been displayed to 0 Then, FOR every line in the open file    IF a desired line (counter hasn’t reached desired number of lines) AND NOT a header line      PRINT the line      SET counter to ADD one    END IF END FOR

## 17.4 Building the code

**Create a new Python script**

### 17.4.1 Usage statement and module importing

The usage statement and module importing from the previous Python exercise module are still accurate for this module, so go ahead and reuse the usage and import statements from the fifth python exercise module within your new Python script.

ANSWER:

#USAGE: python scriptname.py input\_filename [number\_lines\_to\_display]  
import sys

### 17.4.2 SET the input filename

Within your Python script, set the filename variable to be equal to the first command line argument like you did in fourth and fifth Python exercise modules.

ANSWER:

filename = sys.argv[1]

### 17.4.3 IF the user-specified a desired number of lines to display

Within your Python script, set up the conditional statement structure like you did in the fifth Python exercise module to check if the user has provided a desired number of lines to display.

ANSWER:

if len(sys.argv) > 2:

### 17.4.4 THEN SET the desired number of lines

Within your Python script, use the variable assignment statement from the fourth and fifth Python exercise modules to define the desired number of lines as specified by the user, indenting under the conditional statement.

ANSWER:

n\_lines = int(sys.argv[2])

### 17.4.5 OTHERWISE

Use the alternate conditional statement structure from the fifth Python exercise module to start defining the alternate behavior in how we define the desired number of lines variable if the user didn’t specify a specific value. Put this statement in your Python script.

ANSWER:

else:

### 17.4.6 THEN SET the desired number of lines to a default

Use the alternate variable assignment statement from the fifth Python exercise module to define the desired number of lines varialbe if the user didn’t specify a specific value. Put this statement in your Python script, indented correctly.

ANSWER:

n\_lines = 10

### 17.4.7 SET counter of number of lines that have been displayed to 0

Because we no longer assume that the numerical position of the line within the file corresponds to the number of lines that are printed since we’re filtering out the header lines, we need to define a counter variable we can use to track the number of lines that we print. When first initializing this variable, we haven’t printed any lines. Therefore, you want to set the counter variable equal to the value of zero. Do this within your Python script.

ANSWER:

counter = 0

### 17.4.8 FOR every line in the open file

Next, we no longer need to track the position of the line in the f ile, only wanting to work with each line line-by-line in the open file. Therefore, we can go back to the for loop statement that we used in the first Python exercise module. Add the appropriate for loop statement to your Python script.

ANSWER:

for line in open(filename):

### 17.4.9 IF a desired line (because counter hasn’t reached the desired number of lines) and NOT a header line

Let’s break this task down into two separate conditions and then chain the conditions together into one conditional expression.

First, we want to check if the value of the counter is less than the number of desired lines. Write an expression that will evaluate to a boolean variable answering whether an integer is less than another integer using the appropriate variable names.

ANSWER:

counter < n\_lines

Next, we want to verify that the specific line we’re working with is not a header line. Specifically, does the line not start with a pound sign ("#")? Consult the Python notes on negating a boolean expression and checking if a string variable starts with a specific character. Then write an expression that will evaluate to a boolean variable answering whether the line does not start with the pound sign ("#"). Make sure to use the appropriate variable name in the expression.

ANSWER:

not line.startswith("#")

Now, chain the expressions together within your Python script to form a chained conditional expression, within the body of the for loop, indented correctly under your for loop statement. Recall that you want both booleans to be True for the conditional’s body (printing the line) to be executed.

ANSWER:

if counter < n\_lines and not line.startswith("#"):

### 17.4.10 PRINT the line

Finally, reuse the code from the first five Python exercise modules to print the line, adding this print() statement indented correctly under the conditional within your Python script.

ANSWER:

print(line.strip('\r\n'))

### 17.4.11 SET counter to ADD one

If we were to end writing the program now, our script would print every line in the file that wasn’t a header line. Therefore, we have to update the counter every time we print a line so we have an accurate count of the number of lines we’ve printed. Write a variable assignment statement within your Python script that updates the value of the counter variable, adding one to it. Make sure it is appropriately indented in line with the print statement under the conditional statement.

ANSWER:

counter = counter + 1

Within your Python script, you should now have the twelve lines of code together which makes our complete intended goal code – code that prints a user specified number of lines from the beginning of a user specified input file, skipping header lines that begin with a pound sign (#).

**Use the command line to run the Python script and look at the output**

## 17.5 Complete Intended Goal Code

ANSWER:

#USAGE: #USAGE: python scriptname.py input\_filename [number\_lines\_to\_display]  
import sys #import module  
filename = sys.argv[1] #SET the input filename  
if len(sys.argv) > 2: #IF user-specified number of lines provided  
 n\_lines = int(sys.argv[2]) #SET the desired number of lines  
else: #OTHERWISE  
 n\_lines = 10 #SET the desired number of lines to a default  
counter = 0 #SET counter of number of lines that have been displayed ot 0  
for line in open(filename): #FOR every line in the open file  
 if counter < n\_lines and not line.startswith("#"): #IF the counter hasn't reached the max number of desired lines AND NOT a header line  
 print(line.strip('\r\n')) #PRINT the line  
 counter = counter + 1 #SET counter to add one

# 18 Applied Python Exercise 7

Reminder: you will turn in your final script from this module by emailing it to your TA Kate Weaver ([kweave23@jhu.edu](mailto:kweave23@jhu.edu)). Please name it LASTNAME\_tail.py replace “LASTNAME” with your last name.

## 18.1 Goal – Print the desired number of lines from the end of the input file

The goal of this module is to edit your recreated head program from the fifth Python exercise module such that it tail’s an input file. As such, it should print the desired number of lines from the end of the input file, rather than from the beginning.

## 18.2 Learning Objectives

After going through this module, students should be able to:

* State the sub-steps needed to meet the coding goal
* Use the following datatypes, structures, and fundamentals to meet the coding goal:
  + importing modules
  + variable assignment
  + list indexing
  + list length
  + integer conversion
  + opening a file
  + list initialization
  + for loop
  + appending to a list
  + list slicing
  + logical expression
  + conditional statement
  + printing output

## 18.3 Coding Blueprint

We recommend that you start with the final pseudocode from the fifth Python exercise module:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE Then, FOR every line in the open file    IF a desired line (by its numerical position)      PRINT the line    END IF END FOR

Edit this pseudocode such that it reflects what you want to do. Keep lines that are still applicable, remove lines that are no longer applicable, add lines that you need.

Consider that two separate for loops could be useful for this task. One that extracts and stores every line from the file and then one that prints every line from a subset of those that you’ve stored with the first loop.

We’ve provided some pseudocode with blanks that we think will be helpful for you to fill out if you’re unsure of where to start or next steps:

Pseudocode with blanks:

First, we need to SET the input file Next, IF the user-specified a desired number of lines to display    THEN we need to SET the desired number of displayed lines END IF OTHERWISE    THEN we need to SET the desired number of displayed lines to a default END OTHERWISE \_\_\_ a storage list for lines in the file Then, FOR every line in the open file    ~~IF a desired line (by its numerical position)~~      ~~PRINT the line~~    ~~END IF~~    \_\_\_ the line to the storage list END FOR

\_\_\_ a subset of the storage list to be the last \_\_\_\_ items in the storage list \_\_\_ every line in the subset    \_\_\_\_\_ the line END FOR

## 18.4 Building the Code

**Create a new Python script**

Much like how we used the base pseudocode from the fifth Python exercise module, start with the base code that you have from the fifth Python exercise module. Reuse code that is applicable to the task at hand and write new code for the new parts of the program that you specified in the new pseudocode. Consult the learning objectives above to see what data types, structures, and fundamentals we think you will need to build the code. Also, consult the notes on Python fundamentals. Remember to indent when appropriate.

**Use the command line to run the Python script and look at the output**

**Submit your completed script**

# 19 Conclusion: Looking Ahead

## 19.1 Learning Objectives

After going through this module, students should be able to:

* Describe how they will use the Python skills they’ve practiced within these prepwork modules during Quantitative Biology Bootcamp

## 19.2 Looking ahead to next steps

These prepwork modules were intended to give you a brief exposure to the fundamentals of the Python programming language. Within the modules you worked with a .vcf (or variant call format) file that contained data specific to genetic variants. [As explained by this resource](https://www.internationalgenome.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-40/), a.vcf file has a multi-line metadata header, a data header line, and then data lines.

The **metadata header lines** start with ## and provide meta-information specific to the file such as the VCF format version number, when the file was created, what source made the file, if it was quality checked, etc. These lines also may include descriptions of the FILTER, INFO, and FORMAT columns in the data lines. For example, the INFO column likely includes entries for several pieces of information such as the variant’s allele frequency, population specific estimates of the variant’s allele frequency, etc. Each piece of information that is provided in the INFO column will have its own description line in the metadata header.

The **data header line** starts with # and lists at least the 8 required columns for a VCF file. It may include the FORMAT column if genotype data is included in the file. If genotype data is included, it’ll also list the sample/individual IDs.

There are multiple **data lines**. Each data line has information pertaining to a single variant. There is required info about variant location, identity, etc. and, depending on the file, the data lines may or may not contain genotype information for specific samples or individuals. The required columns are

* CHROM – the chromosome where the variant occurs
* POS – the position at which the variant occurs
* ID
* REF – the reference base(s) of the allele
* ALT – the alternate non-reference allele(s) called on at least one o the samples
* QUAL
* FILTER
* INFO – the additional info entries discussed earlier like allele frequency, etc.

If genotype information is provided for samples, these will be included after the FORMAT column, with a column for each included sample.

For the file that you used within these prepwork modules, genotype information was included for 10,000 genetics variants across 2548 samples.

* 0|0 corresponds to a sample being homozygous reference for a variant
* 0|1 or 1|0 correspond to a sample being heterozygous alternate for a variant
* 1|1 corresponds to a sample being homozygous alternate for a variant

In addition to providing each sample’s genotyping information for each variant, the estimated allele frequency of that variant within the population is included.

Using the Python skills that you have practiced and more that you will learn in Quantitative Biology Bootcamp, you will learn to further parse a .vcf file, extracting and visualize specific information like the allele frequencies and genotyptes to make allele frequency histograms and PCAs of the genetic variation:



Figure : Allele Frequency Histogram



Figure : PCA

The skills that you learn throughout Quantitative Biology Bootcamp will help you gain comfort and skills working with many other bioinformatic data files beyond just this .vcf file. Upon completing the course, students should be comfortable using and writing software to work with large-scale biological data. The motivation of this goal is to develop computational and statistical competence in preparation for courses, rotations, thesis research, and careers. Rather than blindly outsourcing bioinformatic components of their work, students will be empowered to understand methodological details and their associated advantages and limitations. This will in turn advance the broader goal of rigor in experimental design, promoting robust and unbiased results.