Programming For Biology 2017

Instructors
Simon Prochnik
Sofia Robb

Table of Contents

            - Backup
            - The Details
        * The Basics
            - Creating a new repository
            - Cloning a Repository
            - Links to Slightly less basic topics
        * Link To Unix 2 Problem Set
- Python
    - Python 1
        * Python Overview
        * Running Python
            - Interactive Interpreter
            - Running Python Scripts
            - Python Script
        * Syntax
            - Python Identifiers
            - Naming conventions for Python Identifiers
            - Reserved Words
            - Lines and Indentation
            - Comments
            - Blank Lines
            - Python Options
        * Data Types and Variables
            - Numbers and Strings
            - Lists
            - Command line parameters: A Special Built-in List
            - Tuple
            - Dictionary
            - Type Conversion
        * Link to Python 1 Problem Set
    - Python 2
        * Operators
            - Arthmetic Operators
            - Assignment Operators
            - Comparison Operators
            - Logical Operators
            - Membership Operators
            - Operator Precedence
        * Truth
            - Use bool() to test for truth
        * Logic: Control Statements
            - If Statement
            - if/elif
        * Numbers
            - integer
            - floating point number
            - complex number
            - Conversion functions
            - Numeric Functions
        * Comparing two numbers
        * Link to Python 2 Problem Set
    - Python 3
        * Sequences
        * What functions go with my object?

    * Strings
        · Quotation Marks
        · Strings and the print() function
        · Errors and Printing
        · Special/Escape Characters
        · Concatenation
        · Determine the length of a string
        · Changing String Case
        · Find and Count
        · Find and Replace
        · Extracting a Substring, or Slicing
        · Locate and Report
        · Other String Methods
    * String Formatting
        · The format() mini-language
        · Summary of special formatting symbols so far
        · What's the point?
    * Lists and Tuples
        · Lists
        · Accessing Values in Lists
        · Changing Values in a List
        · Exracting a Subset of a List, or Slicing
        · List Operators
        · List Functions
        · List Methods
        · Building a List one Value at a Time
    * Link to Python 3 Problem Set
- Python 4
    * Loops
        · while loop
        · While Loop Syntax
        · While/Else
        · For Loops
        · For Loop Syntax
        · For/Else
        · Loop Control
        · Loop Control: Break
        · Loop Control: Continue
        · Iterators
        · List Comprehension
    * Dictionaries
        · Creating a Dictionary
        · Accessing Values in Dictionaries
        · Changing Values in a Dictionary
        · Building a Dictionary one Key/Value at a Time
        · Checking That Dictionary Keys Exist
        · Sorting Dictionary Keys
        · Dictionary Functions
        · Dictionary Methods
    * Sets
        · Set Operators
        · Set Functions
        · Set Methods
    * Link to Python 4 Problem Set

- Python 5
  - I/O and Files
    - Writing to the Screen
    - Reading input from the keyboard
    - Reading from a File
    - Open a File
    - Reading the contents of a file
    - Writing to a File
    - Building a Dictionary from a File
  - Link to Python 5 Problem Set
- Python 6
  - Regular Expressions
    - Individual Characters
    - Character Classes
    - Anchors
    - Quantifiers
    - Variables and Patterns
    - Either Or
    - Subpatterns
    - Using Subpatterns Inside the Regular Expression Match
    - Subpatterns and Greediness
    - Using Subpatterns Outside the Regular Expression Match
    - Practical Example: Codons
    - Truth and Regular Expression Matches
    - Using Regular expressions in substitutions
    - Using subpatterns in the replacement
  - Link to Python 6 Problem Set
- Python 7
  - Functions
    - Creating/Defining a Funtion to Find AT Content:
    - Using/Running/Calling Your function:
    - The details
    - Naming Arguments
    - Keyword Arguments
    - Default Values for Arguments
    - lambda
  - Scope
    - Local Variables
    - Global
- Modules * os.path * os.system * subprocess * Capturing output from a shell pipeline * Capturing output the long way (for a single command) * sys * re * copy * math * random * statistics * glob * argparse
  - Many more modules that do many things
  - Link to Python 7 Problem Set
- Python 8
  - Exception Handling
    - try/except/else/finally
    - Getting more information about an exception
    - Raising an Exception
  - Datastructures
    - Two-demensional lists
    - Lists of dictionaries
    - Dictionaries of lists
    - Dictionaries of dictionaries

        * Link to Python 8 Problem Set
    - Python 9
        * BioPython
            · BioPython Overview
            · BioPython Subtopic 1
            · BioPython Subtopic 2
- Bioinformatic Analysis and Tools
    - Bioinformatic Analysis and Tools Overview
    - Sequence Search and Alignments
    - Assembly
        * DNA
        * RNA
    - NGS
    - Ontology

Programming For Biology 2017

Instructors
Simon Prochnik
Sofia Robb

Big Picture

Why?

*Why is it important for **Biologists** to learn to program?* You probably already know the answer to this question since you are here.

We firmly believe that knowing how to program is just as essential as knowing how to run a gel or set up a PCR reaction. The data we can now get from a single experiment can be overwhelming. This data often needs to be reformatted, filtered, and analyzed in unqiue ways. Programming allows you to perform these tasks in a **reproducible**, **efficient**, and **thoughtful** way.

Helpful Tips

What are our tips for being successful in your efforts to learn to program?

1. Practice, practice, practice. Please, spend as much time possible actually coding.
2. Write only a line or two of code, then test it. If you write too many lines, it becomes more difficult to debug if there is an error.
3. Errors are not failures. Every error you get is a learning opportunity. Every single error you debug is a major success. Fixing errors is how you will cement what you have learned.
4. Don't spend too much time trying to figure out a problem. While it's a great learning experience to try to solve an isssue on your own, it's not fun getting frustrated or spending a lot of time stuck. We are here to help you, so please ask us whenever you need help.
5. Lectures are important, but the practice is more important.
6. Review sessions are important, but practice is more important.

7. It is essential that we help you to learn how to find solutions on your own.

Unix

Unix 1

Unix Overview

What is the Command-Line?

Underlying the pretty Mac OSX GUI is a powerful command-line operating system. The command-line gives you access to the internals of the OS, and is also a convenient way to write custom software and scripts.

Many bioinformatics tools are written to run on the command-line and have no graphical interface. In many cases, a command-line tool is more versatile than a graphical tool, because you can easily combine command-line tools into automated scripts that accomplish tasks without human intervention.

In this course, we will be writing Pyton scripts that are completely command-line based.

The Basics

Logging into Your Workstation

Your workstation is an iMac. To log into it, provide the following information:

Your username: admin

Your password: cshl

Bringing up the Command-Line

To bring up the command-line, use the Finder to navigate to *Applications->Utilities* and double-click on the *Terminal* application. This will bring up a window like the following:

You can open several Terminal windows at once. This is often helpful.

You will be using this application a lot, so I suggest that you drag the Terminal icon into the shortcuts bar at the bottom of your screen.

OK. I've Logged in. What Now?

The terminal window is running **shell** called "bash." The shell is a loop that: 1. Prints a prompt 2. Reads a line of input from the keyboard 3. Parses the line into one or more commands 4. Executes the commands (which usually print some output to the terminal) 5. Go back 1.

There are many different shells with bizarre names like **bash**, **sh**, **csh**, **tcsh**, **ksh**, and **zsh**. The "sh" part means shell. Each shell has different and somewhat confusing features. We have set up your accounts to use **bash**. Stay with **bash** and you'll get used to it, eventually.

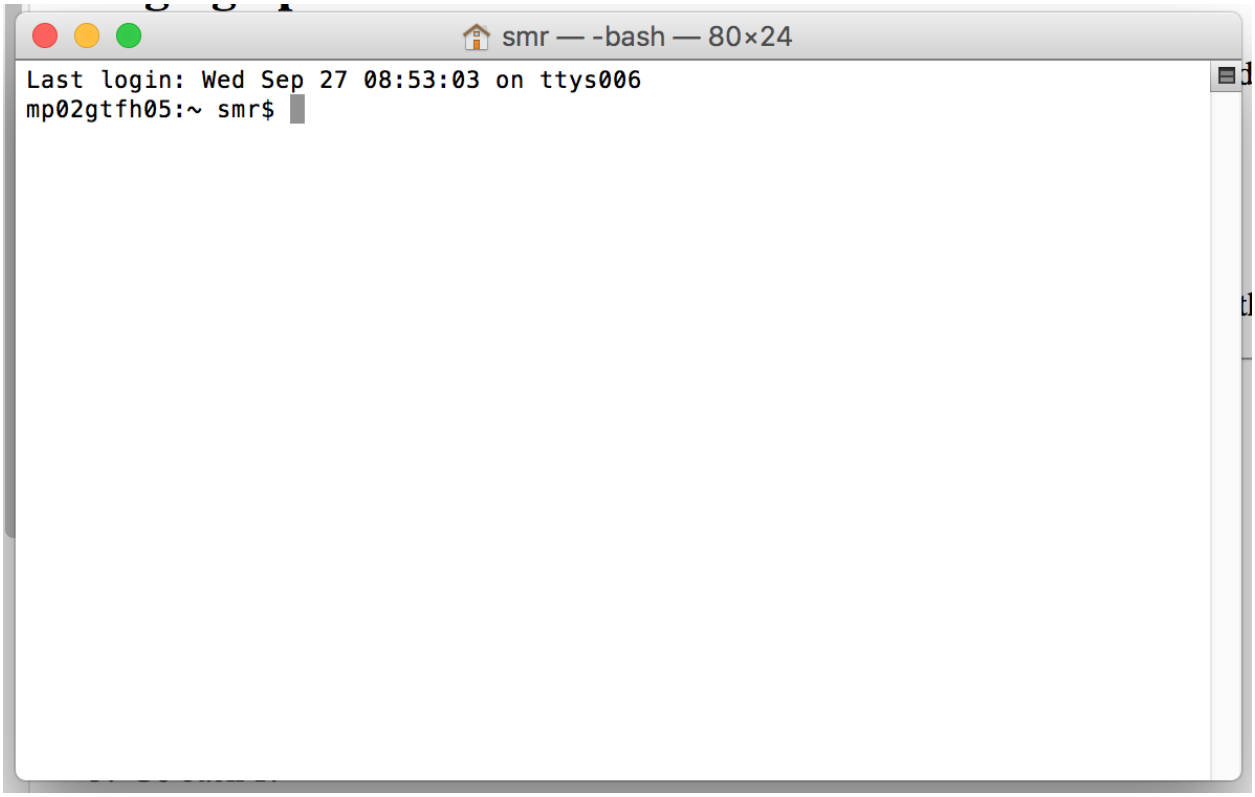

Figure 1: OSX Terminal

Command-Line Prompt

Most of bioinformatics is done with command-line software, so you should take some time to learn to use the shell effectively.

This is a command-line prompt:

```
1 bush202>
```

This is another:

```
1 (~) 51%
```

This is another:

```
1 srobb@bush202 1:12PM>
```

What you get depends on how the system administrator has customized your login. You can customize it yourself when you know how.

The prompt tells you the shell is ready to accept a command. When a long-running command is going, the prompt will not reappear until the system is ready to deal with your next request.

Issuing Commands

Type in a command and press the <Enter> key. If the command has output, it will appear on the screen. Example:

```
(~) 53% ls -F
GNUstep/                  cool_elegans.movies.txt  man/
INBOX                     docs/                    mtv/
INBOX~                    etc/                     nsmail/
Mail@                     games/                   pcod/
News/                     get_this_book.txt        projects/
axhome/                   jcod/                    public_html/
bin/                      lib/                     src/
build/                    linux/                   tmp/
ccod/
(~) 54%
```

The command here is `ls -F` which produces a listing of files and directories in the current directory (more on that later). Below its output, the command prompt appears again.

Some programs will take a long time to run. After you issue their command names, you won't recover the shell prompt until they're done. You can either launch a new shell (from Terminal's File menu), or run the command in the background by adding an ampersand after the command

```
(~) 54% long_running_application &
(~) 55%
```

> The command will now run in the background until it is finished. If it has any output, the output will be printed to the terminal window. You may wish to capture the output in a file (called redirection). We'll describe this later.

Command-Line Editing

Most shells offer command-line editing. Up until the comment you press , you can go back over the command-line and edit it using the keyboard. Here are the most useful keystrokes:

- *Backspace*: Delete the previous character and back up one.
- *Left arrow*, right arrow: Move the text insertion point (cursor) one character to the left or right.
- *control-a (^a)*: Move the cursor to the beginning of the line. (Mnemonic: A is first letter of alphabet)
- *control-e (^e)*: Move the cursor to the end of the line. Mnemonic: E for the End (^z was already used for interrupt a command).
- *control-d (^d)*: Delete the character currently under the cursor. D=Delete.
- *control-k (^k)*: Delete the entire line from the cursor to the end. k=kill. The line isn't actually deleted, but put into a temporary holding place called the "kill buffer". This is like cutting text
- *control-y (^y)*: Paste the contents of the kill buffer onto the command-line starting at the cursor. y=yank. This is like paste.
- *Up arrow, down arrow*: Move up and down in the command history. This lets you reissue previous commands, possibly after modifying them.

There are also some useful shell commands you can issue:

- `history` Show all the commands that you have issued recently, nicely numbered.
- `!<number>` Reissue an old command, based on its number (which you can get from `history`).
- `!!` Reissue the immediate previous command.
- `!<partial command string>`: Reissue the previous command that began with the indicated letters. For example, `!l` (the letter el, not a number 1) would reissue the`ls -F` command from the earlier example.

bash offers automatic command completion and spelling correction. If you type part of a command and then the tab key, it will prompt you with all the possible completions of the command. For example:

```
1 (~) 51% fd<tab><tab>
2 (~) 51% fd
3 fd2ps    fdesign  fdformat fdlist   fdmount  fdmountd fdrawcmd
     fdumount
4 (~) 51%
```

> If you hit tab after typing a command, but before pressing <Enter>, **bash** will prompt you with a list of file names. This is because many commands operate on files.

Wildcards

You can use wildcards when referring to files. `*` stands for zero or more characters. `?` stands for any single character. For example, to list all files with the extension ".txt", run `ls` with the wildcard pattern "*.txt"

```
1 (~) 56% ls -F *.txt
2 final_exam_questions.txt  genomics_problem.txt
3 genebridge.txt            mapping_run.txt
```

There are several more advanced types of wildcard patterns that you can read about in the **tcsh** manual page. For example, if you want to match files that begin with the characters "f" or "g" and end with ".txt", you can use a range of characters inside square brackets `[f-g]` as part of the wildcard pattern. Here's an example

```
1 (~) 57% ls -F [f-g]*.txt
2 final_exam_questions.txt  genebridge.txt
     genomics_problem.txt
```

Home Sweet Home

When you first log in, you'll be placed in a part of the system that is your personal directory, called the *home directory*. You are free to do with this area what you will: in particular you can create and delete files and other directories. In general, you cannot create files elsewhere in the system.

Your home directory lives somewhere in the filesystem. On our iMacs, it is a directory with the same name as your login name, located in `/Users`. The full directory path is therefore `/Users/username`. Since this is a pain to write, the shell allows you to abbreviate it as `~username` (where "username" is your user name), or

simply as `~`. The weird character (called "tilde" or "twiddle") is usually hidden at the upper left corner of your keyboard.

To see what is in your home directory, issue the command `ls -F`:

```
(~) % ls -F
INBOX          Mail/          News/          nsmail/        public_html/
```

This shows one file "INBOX" and four directories ("Mail", "News") and so on. (The `-F` in the command turns on fancy mode, which appends special characters to directory listings to tell you more about what you're seeing. `/` at the end of a filename means that file is a directory.)

In addition to the files and directories shown with `ls -F`, there may be one or more hidden files. These are files and directories whose names start with a `.` (called the "dot" character). To see these hidden files, add an `a` to the options sent to the `ls` command:

```
(~) % ls -aF
./                .cshrc            .login            Mail/
../               .fetchhost        .netscape/        News/
.Xauthority       .fvwmrc           .xinitrc*         nsmail/
.Xdefaults        .history          .xsession@        public_html/
.bash_profile     .less             .xsession-errors
.bashrc           .lessrc           INBOX
```

> Whoa! There's a lot of hidden stuff there. But don't go deleting dot files. Many of them are essential configuration files for commands and other programs. For example, the `.profile` file contains configuration information for the **bash** shell. You can peek into it and see all of **bash**'s many options. You can edit it (when you know what you're doing) in order to change things like the command prompt and command search path.

Getting Around

You can move around from directory to directory using the `cd` command. Give the name of the directory you want to move to, or give no name to move back to your home directory. Use the `pwd` command to see where you are (or rely on the prompt, if configured):

```
(~/docs/grad_course/i) 56% cd
(~) 57% cd /
(/) 58% ls -F
bin/          dosc/         gmon.out      mnt/          sbin/
boot/         etc/          home@         net/          tmp/
cdrom/        fastboot      lib/          proc/         usr/
dev/          floppy/       lost+found/   root/         var/
(/) 59% <b>cd ~/docs/</b>
(~/docs) 60% <b>pwd</b>
/usr/home/lstein/docs
(~/docs) 62% cd ../projects/
(~/projects) 63% ls
Ace-browser/                bass.patch
Ace-perl/                   cgi/
Foo/                        cgi3/
Interface/                  computertalk/
Net-Interface-0.02/         crypt-cbc.patch
Net-Interface-0.02.tar.gz   fixer/
Pts/                        fixer.tcsh
Pts.bak/                    introspect.pl*
PubMed/                     introspection.pm
SNPdb/                      rhmap/
Tie-DBI/                    sbox/
ace/                        sbox-1.00/
atir/                       sbox-1.00.tgz
bass-1.30a/                 zhmapper.tar.gz
bass-1.30a.tar.gz
(~/projects) 64%
```

Each directory contains two special hidden directories named `.` and `...` The first, `.` refers always to the current directory. `..` refers to the parent directory. This lets you move upward in the directory hierarchy like this:

```
(~/docs) 64% cd ..
```

and to do arbitrarily weird things like this:

```
(~/docs) 65% cd ../../lstein/docs
```

The latter command moves upward two levels, and then into a directory named `docs` inside a directory called `lstein`.

If you get lost, the `pwd` command prints out the full path to the current directory:

```
(~) 56% pwd
/Users/lstein
```

Essential Unix Commands

With the exception of a few commands that are built directly into the shell, all Unix commands are standalone executable programs. When you type the name of a command, the shell will search through all the directories listed in the PATH environment variable for an executable of the same name. If found, the shell will execute the command. Otherwise, it will give a "command not found" error.

Most commands live in `/bin`, `/usr/bin`, or `/usr/local/bin`.

Getting Information About Commands

The `man` command will give a brief synopsis of a command. Let's get information about the command `wc`

```
(~) 76% man wc
Formatting page, please wait...
WC(1)                                                       WC(1)

NAME
       wc - print the number of bytes, words, and lines in files

SYNOPSIS
       wc [-clw] [--bytes] [--chars] [--lines] [--words] [--help]
       [--version] [file...]

DESCRIPTION
       This manual page documents the  GNU  version  of  wc.   wc
       counts  the  number  of bytes, whitespace-separated words,
...
```

Finding Out What Commands are on Your Computer

The `apropos` command will search for commands matching a keyword or phrase. Here's an example that looks for commands related to 'column'

```
(~) 100% apropos column
showtable (1)        - Show data in nicely formatted columns
colrm (1)            - remove columns from a file
column (1)           - columnate lists
fix132x43 (1)        - fix problems with certain (132 column)
   graphics
modes
```

Arguments and Command Switches

Many commands take arguments. Arguments are often the names of one or more files to operate on. Most commands also take command-line "switches" or "options", which fine-tune what the command does. Some commands recognize "short switches" that consist of a minus sign `-` followed by a single character, while others recognize "long switches" consisting of two minus signs `--` followed by a whole word.

The `wc` (word count) program is an example of a command that recognizes both long and short options. You can pass it the `-c`, `-w` and/or `-l` options to count the characters, words and lines in a text file, respectively. Or you can use the longer but more readable, `--chars`, `--words` or `--lines` options. Both these examples count the number of characters and lines in the text file `/var/log/messages`:

```
(~) 102% wc -c -l /var/log/messages
     23     941 /var/log/messages
(~) 103% wc --chars --lines /var/log/messages
     23     941 /var/log/messages
```

You can cluster short switches by concatenating them together, as shown in this example:

```
(~) 104% wc -cl /var/log/messages
     23     941 /var/log/messages
```

Many commands will give a brief usage summary when you call them with the `-h` or `--help` switch.

Spaces and Funny Characters

The shell uses whitespace (spaces, tabs and other non-printing characters) to separate arguments. If you want to embed whitespace in an argument, put single quotes around it. For example:

```
mail -s 'An important message' 'Bob Ghost <bob@ghost.org>'
```

> This will send an e-mail to the fictitious person Bob Ghost. The `-s` switch takes an argument, which is the subject line for the e-mail. Because the desired subject contains spaces, it has to have quotes around it. Likewise, my name and e-mail address, which contains embedded spaces, must also be quoted in this way.

Certain special non-printing characters have *escape codes* associated with them:

Escape Code	Description
\n	new line character
\t	tab character
\r	carriage return character
\a	bell character (ding! ding!)
\nnn	the character whose ASCII code is **nnn**

Useful Commands

Here are some commands that are used extremely frequently. Use `man` to learn more about them. Some of these commands may be useful for solving the problem set ;-)

Manipulating Directories

Command	Description
`ls`	Directory listing. Most frequently used as `ls -F` (decorated listing), `ls -l` (long listing), `ls -a` (list all files).
`mv`	Rename or move a file or directory.
`cp`	Copy a file.
`rm`	Remove (delete) a file.
`mkdir`	Make a directory
`rmdir`	Remove a directory
`ln`	Create a symbolic or hard link.
`chmod`	Change the permissions of a file or directory.

Command	Description
`cat`	Concatenate program. Can be used to concatenate multiple files together into a single file, or, much more frequently, to view the contents of a file or files in the terminal.
`more`	Scroll through a file page by page. Very useful when viewing large files. Works even with files that are too big to be opened by a text editor.
`less`	A version of `more` with more features.
`head`	View the first few lines of a file. You can control how many lines to view.
`tail`	View the end of a file. You can control how many lines to view. You can also use `tail -f` to view a file that you are writing to.
`wc`	Count words, lines and/or characters in one or more files.
`tr`	Substitute one character for another. Also useful for deleting characters.
`sort`	Sort the lines in a file alphabetically or numerically.
`uniq`	Remove duplicated lines in a file.
`cut`	Remove sections from each line of a file or files.
`fold`	Wrap each input line to fit in a specified width.
`grep`	Filter a file for lines matching a specified pattern. Can also be reversed to print out lines that don't match the specified pattern.
`gzip` (`gunzip`)	Compress (uncompress) a file.
`tar`	Archive or unarchive an entire directory into a single file.
`emacs`	Run the Emacs text editor (good for experts).
`vi`	Run the vi text editor (better for experts).

Networking

Command	Description

Command	Description
`ssh`	A secure (encrypted) way to log into machines.
`scp`	A secure way to copy (cp) files to and from remote machines.
`ping`	See if a remote host is up.
`ftp`/ `sftp` (secure)	transfer files using the File Transfer Protocol.

Standard I/O and Command Redirection

Unix commands communicate via the command-line interface. They can print information out to the terminal for you to see, and accept input from the keyboard (that is, from *you*!)

Every Unix program starts out with three connections to the outside world. These connections are called "streams", because they act like a stream of information (metaphorically speaking):

Stream Type	Description
standard input	This is a communications stream initially attached to the keyboard. When the program reads from standard input, it reads whatever text you type in.
standard output	This stream is initially attached to the command window. Anything the program prints to this channel appears in your terminal window.
standard error	This stream is also initially attached to the command window. It is a separate channel intended for printing error messages.

The word "initially" might lead you to think that standard input, output and error can somehow be detached from their starting places and reattached somewhere else. And you'd be right. You can attach one or more of these three streams to a file, a device, or even to another program. This sounds esoteric, but it is actually very useful.

A Simple Example

The `wc` program counts lines, characters and words in data sent to its standard input. You can use it interactively like this:

```
1 (~) 62% wc
2 Mary had a little lamb,
3 little lamb,
4 little lamb.
5
6 Mary had a little lamb,
7 whose fleece was white as snow.
8 ^D
9       6      20     107
```

In this example, I ran the `wc` program. It waited for me to type in a little poem. When I was done, I typed the END-OF-FILE character, control-d (^d for short). `wc` then printed out three numbers indicating the

number of lines, words and characters in the input.

More often, you'll want to count the number of lines in a big file; say a file filled with DNA sequences. You can do this by *redirecting* the contents of a file to the standard input of `wc`. This uses the `<` symbol:

```
(~) 63% wc < big_file.fasta
      2943    2998     419272
```

If you wanted to record these counts for posterity, you could redirect standard output as well using the `>` symbol:

```
(~) 64% wc < big_file.fasta > count.txt
```

Now if you `cat` the file *count.txt*, you'll see that the data has been recorded. `cat` works by taking its standard input and copying it to standard output. We redirect standard input from the *count.txt* file, and leave standard output at its default, attached to the terminal:

```
(~) 65% cat < count.txt
      2943    2998     419272
```

Redirection Meta-Characters

Here's the complete list of redirection commands for `bash`:

Redirect command	Description
`< myfile.txt`	Redirect standard input to file
`> myfile.txt`	Redirect standard output to file
`1 > myfile.txt`	Redirect just standard output to file (same as above)
`2 > myfile.txt`	Redirect just standard error to file
`> myfile.txt 2>&1`	Redirect both stdout and stderr to file

These can be combined. For example, this command redirects standard input from the file named `/etc/passwd`, writes its results into the file `search.out`, and writes its error messages (if any) into a file named `search.err`. What does it do? It searches the password file for a user named "root" and returns all lines that refer to that user.

```
(~) 66% grep root < /etc/passwd > search.out 2> search.err
```

Filters, Filenames and Standard Input

Many Unix commands act as filters, taking data from a file or standard input, transforming the data, and writing the results to standard output. Most filters are designed so that if they are called with one or more filenames on the command-line, they will use those files as input. Otherwise they will act on standard input. For example, these two commands are equivalent:

```
(~) 66% grep 'gatttgc' < big_file.fasta
(~) 67% grep 'gatttgc' big_file.fasta
```

Both commands use the `grep` command to search for the string "gatttgc" in the file `big_file.fasta`. The first one searches standard input, which happens to be redirected from the file. The second command is explicitly given the name of the file on the command line.

Sometimes you want a filter to act on a series of files, one of which happens to be standard input. Many commands let you use `-` on the command-line as an alias for standard input. Example:

```
(~) 68% grep 'gatttgc' big_file.fasta bigger_file.fasta -
```

This example searches for "gatttgc" in three places. First it looks in file `big_file.fasta`, then in `bigger_file.fasta`, and lastly in standard input (which, since it isn't redirected, will come from the keyboard).

Standard I/O and Pipes

The coolest thing about the Unix shell is its ability to chain commands together into pipelines. Here's an example:

```
(~) 65% grep gatttgc big_file.fasta | wc -l
22
```

There are two commands here. `grep` searches a file or standard input for lines containing a particular string. Lines which contain the string are printed to standard output. `wc -l` is the familiar word count program, which counts words, lines and characters in a file or standard input. The `-l` command-line option instructs `wc` to print out just the line count. The `|` character, which is known as a "pipe", connects the two commands together so that the standard output of `grep` becomes the standard input of `wc`. Think of pipes connecting streams of data flowing.

What does this pipe do? It prints out the number of lines in which the string "gatttgc" appears in the file `big_file.fasta`.

More Pipe Idioms

Pipes are very powerful. Here are some common command-line idioms.

Count the Number of Times a Pattern does NOT Appear in a File

The example at the top of this section showed you how to count the number of lines in which a particular string pattern appears in a file. What if you want to count the number of lines in which a pattern does **not** appear?

Simple. Reverse the test with the `-v` switch:

```
(~) 65% grep -v gatttgc big_file.fasta | wc -l
2921
```

Uniquify Lines in a File

If you have a long list of names in a text file, and you want to weed out the duplicates:

```
(~) 66% sort long_file.txt | uniq > unique.out
```

This works by sorting all the lines alphabetically and piping the result to the `uniq` program, which removes duplicate lines that occur one after another. That's why you need to sort first. The output is placed in a file named `unique.out`.

Concatenate Several Lists and Remove Duplicates

If you have several lists that might contain repeated entries among them, you can combine them into a single unique list by concatenating them together, then sorting and uniquifying them as before:

```
(~) 67% cat file1 file2 file3 file4 | sort | uniq
```

Count Unique Lines in a File

If you just want to know how many unique lines there are in the file, add a `wc` to the end of the pipe:

```
(~) 68% sort long_file.txt | uniq | wc -l
```

Page Through a Really Long Directory Listing

Pipe the output of `ls` to the `more` program, which shows a page at a time. If you have it, the `less` program is even better:

```
(~) 69% ls -l | more
```

Monitor a Rapidly Growing File for a Pattern

Pipe the output of `tail -f` (which monitors a growing file and prints out the new lines) to `grep`. For example, this will monitor the `/var/log/syslog`file for the appearance of e-mails addressed to 'mzhang':

```
(~) 70% tail -f /var/log/syslog | grep mzhang
```

Advanced Unix

Here are a few more advanced Unix commands that are very useful and when you have time you should investigate further. We list the page numbers in the Internet Version (v3) of 'The Linux Command Line' by William Shotts.

- `awk`
- `sed` (p.295)
- `perl` one-liners
- `for` loops (p. 453)

Link to Unix 1 Problem Set

Link To Unix 2 Problem Set

Python

Python 1

Python Overview

Python has - data types - functions - objects - classes - methods

Data types are just different types of data which are discussed in more detail later. Examples of data types are integer numbers and strings of letters and numbers (text). These can be stored in variables.

Funtions do something with data, such as a calculation. Some functions are already built into Python. You can create your own functions as well.

Objects are a way of grouping a set of data and functions (methods) that act on that data

Classes are a way to encapulate (organize) variables and functions. Objects get their variables and methods from the class they belong to.

Methods are just functions that belong to a Class. Objects that belong to the a Class can use Methods from that Class.

Running Python

Interactive Interpreter

Python can be run one line at a time in an interactive interpreter.
To lauch the interpreter type the following into your terminal window:

```
$ python
```
Note: '$' indicates the command line prompt

First Python Commands:

```
$ python
```

```
>>> print("Hello, PFB2017!")
Hello, PFB2017!
```

> Note: `print` is a function. Function names are followed by (), so formally, the function is `print()`

Running Python Scripts

Typing the python command followed by the name of a script makes python execute the script. Recall that we just saw you can run an interactive interpreter by just typing `python` on the command line

Python Script

- The same code from above is typed into a text file.
- Python scripts are always saved in files whose names have the extension '.py' (i.e. the filename ends with '.py').

File Contents:

```
#!/usr/bin/python3
print ("Hello, PFB2017!")
```

Execute the Python script:

```
$ python test.py
```

This produces the following result:

```
Hello, PFB2017!
```

Syntax

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, seq_id and seq_ID are two different identifiers in Python.

Naming conventions for Python Identifiers

- The first character is lowercase, unless it is a name of a Class. Classes should begin with an uppercase characters. (ex. Seq)
- Private identifiers begin with an underscore. (ex. `_private`)
- Strong private identifiers begin with two underscores. (ex. `__private`)
- Language-defined special names begin and end with two underscores. (ex. `__special__`)

Reserved Words

The following is a list of Python keywords. These are special words that already have a purpose in python and therefore cannot be used in indentifier names.

```
and          exec         not
as           finally      or
assert       for          pass
break        from         print
class        global       raise
continue     if           return
def          import       try
del          in           while
elif         is           with
else         lambda       yield
except
```

Lines and Indentation

Python denotes blocks of code by line indentation. Incorrect line spacing and/or indention will cause an error to be reported or could make your code run in a way you don't expect. You can get help with indentation from good text editors or IDEs.

The number of spaces in the indentation need to be consistent but a specific number is not required. All lines of code, or statements, within a single block must be indented in the same way. For example:

Comments

Comments are an essential programming practice. Making a note of what a line or block of code is doing will help the writer and readers of the code. This includes you!

#

Comments start after a pound or hash symbol. All characters after the #, up to the end of the line are part of the comment and are ignored by python.

The first line of a script starting with `#!` is a special example of a comment that also has the special function in unix of telling the unix shell how to run the script.

```
#!/usr/bin/python3

# this is my first script
print ("Hello, PFB2017!") # this line prints
```

Blank Lines

Blank lines are also important for increasing the readability of the code. Blank lines are ignored by the python interpreptor

Python Options

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser (also PYTHONDEBUG=x)
-E     : ignore environment variables (such as PYTHONPATH)
-h     : print this help message and exit
```

Data Types and Variables

sep note: overview of all types: start with constants: numbers, strings ‘’ “” ’’’’’ r” = raw strings, + for concat, * for repeat iterable, string, int, float, tuple, list, dictionary, set

This is our first look at variables and data types. Each data type will be discussed in more detail in subsequent sections.

The first concept to consider is that python data types are either immutable (unchangeable) or not. Literal numbers, strings and tuples cannot be changed. Lists, dictionaries and sets can be changed. So can individual (scalar) variables. You can store data in memory by putting it in different kinds variables. You use the = sign to assign a value to a variable.

Numbers and Strings

Numbers and strings are two data types. Literal numbers and strings like this `5` or `'my name is'` are immutable. However, their values can be stored in variables and then changed.

For Example:

```
first_variable = 5
```

Different types of data can be assigned to variables, i.e., integers (1,2,3), floats (floating point numbers, 3.1415), and strings (text).

For Example:

```
count   = 10      # this is an integer
average = 2.5     # this is a float
message = "Welcome to python" # this is a string
```

10, 2.5, and "Welcome to python" are singular pieces of data being stored in an indivudual variables.

Collections of data can also be stored in special data types, i.e., tuples, lists, sets, and dictionaries.

Lists

- Lists are used to store an ordered, *indexed* collection of data.
- Lists are mutable: the number of elements in the list and what's stored in each element can change
- Lists are enclosed in square brackets and items are separated by commas

```
[ 'atg' , 'aaa' , 'agg' ]
```

Index	Value
0	atg
1	aaa
2	agg

Command line parameters: A Special Built-in List

Command line parameters follow the name of a script or program and have spaces between them. They allow a user to pass information to a script on the command line when that script is being run. Python stores all the pieces of the command line in a special list called sys.argv.

You need to import the sys module at the beginning of your script like this

```
import sys
```

If you write this on the command line:

```
$ calculate_sum.py 5 7
```

This happens inside the script: > the script name, and the numbers 5 and 7 are contained in a list called `sys.argv`.

> These are the command line parameters, or arguments you want to pass to your script.
> `sys.argv[0]` is the script name.
> You can access values of the other parameters by their indices, starting with 1, so `sys.argv[1]` is 5 and `sys.argv[2]` is 7.

If you wanted to calculate the sum in your script, you would add these two variables and print the result. Maybe your code would look something like this

```
#!/usr/bin/python3
import sys
a = sys.argv[1]
b = sys.argv[2]
print(a+b) # + is a sum operator on integers
```

Tuple

- Tuples are similar to lists and contain ordered, *indexed* collection of data.
- **Tuples are immutable: you can't change the values or the number of values**
- A tuple is enclosed in parenteses and items are separated by commas.

```
( 'Jan' , 'Feb' , 'Mar' , 'Apr' , 'May' , 'Jun' , 'Jul' , 'Aug'
   , 'Sep' , 'Oct' , 'Nov' , 'Dec' )
```

Index	Value
0	Jan
1	Feb
2	Mar
3	Apr
4	May
5	Jun
6	Jul
7	Aug
8	Sep
9	Oct
10	Nov
11	Dec

Dictionary

- Dictionaries are good for storing data that can be represented as a two-column table.
- They store unordered collections of key/value pairs.
- A dictionary is enclosed in curly braces. and sets of Key/Value pairs are separated by commas
- A colon is written between each key and value. Commas separate key:value pairs.

```
{ 'TP53' :
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
    , 'BRCA1' :
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
    }
```

Key	Value
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAG
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCG

Type Conversion

Sometimes you may need to convert data into a specific type. Here are some examples of functions that will help you to do this.

Function	Description
int(x)	Converts x to an integer.
float(x)	Converts x to a floating-point number.
str(x)	Converts x to a string.
chr(x)	Converts an integer to a character.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary. d must be a sequence of (key,value) tuples.
repr()	Makes a string representation of an object. Useful for finding out what kind (Class) of object you are dealing with.

Link to Python 1 Problem Set

Python 2

Operators

Arthmetic Operators

Operator	Description	Example	Result
+	Addition	3+2	5
-	Subtraction	3-2	1
*	Multiplication	3*2	6
/	Division	3/2	1.5

Operator	Description	Example	Result
%	Modulus (divides left operand by right operand and returns the remainder)	3%2	1
**	Exponent	3**2	9
//	Floor Division (result is the quotient with digits after the decimal point removed. If one of the operands is negative, the result is floored, i.e., rounded away from zero	3//2 ; -11//3	1 ; -4

Assignment Operators

Operator	Equivalent to	Example	result evaluates to
=	a = 3	result = 3	3
+=	result = result + 2	result = 3 ; result += 2	5
-=	result = result - 2	result = 3 ; result -= 2	1
=	result = result * 2	result = 3 ; result *= 2	6
/=	result = result / 2	result = 3 ; result /= 2	1.5
%=	result = result % 2	result = 3 ; result %= 2	1
=	result = result ** 2	result = 3 ; result **= 2	9
//=	result = result // 2	result = 3 ; result //= 3	1

Comparison Operators

These operators compare two values and returns true or false.

Operator	Description	Example	Result
==	equal to	3 == 2	False
!=	not equal	3 != 2	True
>	greater than	3 > 2	True
<	less than	3 < 2	False
>=	greater than or equal	3 >= 2	True
<=	less than or equal	3 <= 2	False

Logical Operators

Operator	Description	Example	Result
and	True if left operand is True and right operand is True	bool(3>=2 and 2<3)	True
or	TRUE if left operand is Treu or right operand is True	bool(3==2 or 2<3)	True
not	Reverses the logical status	bool(not False)	True

Membership Operators

Operator	Description
in	True if a value is included in a list, tuple, or string
not in	True if a value is absent in a list, tuple, or string

For Example:

```
>>> dna =
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> 'TCT' in dna
True
>>>
>>> 'ATG' in dna
False
>>> 'ATG' not in dna
True
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> 'atg' in codons
True
>>> 'ttt' in codons
False
```

Operator Precedence

Operators are listed in order of precedence. Highest listed first. Not all the operators listed here are mentioned above.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive 'OR' and regular 'OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is	Identity operator
is not	Non-identity operator
in	Membership operator
not in	Negative membership operator
not, or, and	logical operators

Truth

Lets take a step back, What is truth?

Everything is true, except for:

expression	TRUE/FALSE
`0`	FALSE
`None`	FALSE
`False`	FALSE

expression	TRUE/FALSE
`''` (empty string)	FALSE
`[]` (empty list)	FALSE
`()` (empty tuple)	FALSE
`{}` (empty dictionary)	FALSE

Which means that these are True:

expression	TRUE/FALSE
`'0'`	TRUE
`'None'`	TRUE
`'False'`	TRUE
`'True'`	TRUE
`' '` (string of one blank space)	TRUE

Use `bool()` to test for truth

```
>>> bool(True)
True
>>> bool('True')
True
>>>
>>>
>>> bool(False)
False
>>> bool('False')
True
>>>
>>>
>>> bool(0)
False
>>> bool('0')
True
>>>
>>>
>>> bool('')
False
>>> bool(' ')
True
>>>
>>>
>>> bool(())
False
>>> bool([])
False
>>> bool({})
False
```

Logic: Control Statements

Control Statements are used to direct the flow of your code and create the oportunity for decision making. Control statements foundation is build on truth.

If Statement

- Use the If Statement to test for truth and to execute lines of code if true.
- When the expression evaluates to true each of the statements indented below the if statment, also known as the nested statement block, will be executed.

IF

```
if expression :
  statement
  statement
```

For Example:

```
dna =
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'AGC' in dna:
  print('found AGC in your dna sequence')
```

Returns:

```
found AGC in your dna sequence
```

ELSE

- The If portion of the if/else statement behave as before.
- The first indented block is executed if the condition is true.
- If the condition is false, the second indented else block is executed.

```
dna =
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
if 'ATG' in dna:
  print('found ATG in your dna sequence')
else:
  print('did not find ATG in your dna sequence')
```

Returns:

```
did not find ATG in your dna sequence
```

if/elif

- The if condition is tested as before and the indented block is executed if the condition is true.
- If it's false, the indented block following the elif is executed if the first elif condition is true.
- Any remaining elif conditions will be tested in order until one is found to be true. If none is true, the else indented block is executed.

```
count = 60
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
60 is greater than 50
```

Let's change count to 20, which statement block gets executed?

```
count = 20
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
20 is less than 100
```

What happens when count is 50?

```
count = 50
if count < 0:
  message = "is less than 0"
  print(count, message)
elif count < 50:
  message = "is less than 50"
  print (count, message)
elif count > 50:
  message = "is greater than 50"
  print (count, message)
else:
  message = "must be 50"
  print(count, message)
```

Returns:

```
50 must be 50
```

Numbers

Python recognizes 3 types of numbers: integers, float point numbers, and complex numbers.

integer

- known as an int
- an int can be positve or negative
- and **does not** contain a decimal point.

floating point number

- known as a float
- a floating point number can be positve or negative
- and **does** contain a decimal point

complex number

- known as complex
- is in the form of a+bi where i is the imaginary part.

Conversion functions

function	Description
int(x)	to convert x to a plain integer
float(x)	to convert x to a floating-point number
complex(x)	to convert x to a complex number with real part x and imaginary part zero

function	Description
complex(x, y)	to convert x and y to a complex number with real part x and imaginary part y

```
>>> int(2.3)
2
>>> float(2)
2.0
>>> complex(2.3)
(2.3+0j)
>>> complex(2.3,2)
(2.3+2j)
```

Numeric Functions

function	Description
abs(x)	The absolute value of x: the (positive) distance between x and zero.
round(x [,n])	x rounded to n digits from the decimal point. round(0.5) is 1.0 and round(-0.5) is -1.0.
max(x1, x2,...)	The largest positive argument is returned
min(x1, x2,...)	The smallest argument is returned

```
>>> abs(2.3)
2.3
>>> abs(-2.9)
2.9
>>> round(2.3)
2
>>> round(2.5)
2
>>> round(2.9)
3
>>> round(-2.9)
-3
>>> round(-2.3)
-2
>>> round(-2.009,2)
-2.01
>>> max(4,-5,5,1,11)
11
>>> min(4,-5,5,1,11)
-5
```

Many numeric functions are not built into the Python core and need to be included in our script if we want

to use them. To include them at the tip of the script type: `import math`

These next functions are found in the math module and need to be imported. To use these function, prepend the function with the module name, i.e, `math.ceil(15.5)`

math.function	Description
ceil(x)	The smallest integer not greater than x is returned
floor(x)	the largest integer not greater than x is returned.
exp(x)	The exponential of x: ex is returned
log(x)	the natural logarithm of x, for x > 0 is returned
log10(x)	The base-10 logarithm of x for x > 0 is returned
modf(x)	The fractional and integer parts of x are returned in a two-item tuple.
pow(x, y)	The value of x**y is returned
sqrt(x)	The square root of x for x > 0 is returned

```
>>> import math
>>>
>>> math.ceil(2.3)
3
>>> math.ceil(2.9)
3
>>> math.ceil(-2.9)
-2
>>> math.floor(2.3)
2
>>> math.floor(2.9)
2
>>> math.floor(-2.9)
-3
>>> math.exp(2.3)
9.974182454814718
>>> math.exp(2.9)
18.17414536944306
>>> math.exp(-2.9)
0.05502322005640723
>>>
>>> math.log(2.3)
0.8329091229351039
>>> math.log(2.9)
1.0647107369924282
>>> math.log(-2.9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
>>> math.log10(2.3)
0.36172783601759284
>>> math.log10(2.9)
0.4623979978989561
>>>
>>> math.modf(2.3)
(0.2999999999999998, 2.0)
>>>
>>> math.pow(2.3,1)
2.3
>>> math.pow(2.3,2)
5.289999999999999
>>> math.pow(-2.3,2)
5.289999999999999
>>> math.pow(2.3,-2)
0.18903591682419663
>>>
>>> math.sqrt(25)
5.0
>>> math.sqrt(2.3)
1.51657508881031
>>> math.sqrt(2.9)
1.70293863659264
```

Comparing two numbers

The simple function `cmp(x,y)` is no longer available in python3.

Use this idiom instead:

```
cmp = (x>y)-(x<y)
```

It returns three different values depending on x and y

- if x<y -1 is returned
- if x>y 1 is returned
- x == y 0 is returned

Link to Python 2 Problem Set

Python 3

Sequences

In the next section, we will learn about strings, tuples and lists. These are all examples of python sequences. A string is a sequence of characters `'ACGTGA'`, a tuple `(0.23, 9.74, -8.17, 3.24, 0.16)` and a list `['dog', 'cat', 'bird']` are sequences of any kind of data. We'll see much more detail in a bit.

In python a type of object gets operations that belong to that type. Sequences have sequence operations so strings can also use sequence operations. Strings also have their own specific operations.

You can ask what the length of any sequence is

```
>>>len('ACGTGA') # length of a string
6
>>>len ( (0.23, 9.74, -8.17, 3.24, 0.16) )   # length of a tuple,
    needs two parentheses (( ))
5
>>>len(['dog', 'cat', 'bird'])  # length of a list
3
```

You can also use string-specific functions on strings, but not on lists and vice versa. We'll learn a lot more about this later on. `rstrip()` is a string method or function. You get an error if you try to use it on a list.

```
>>> 'ACGTGA'.rstrip('A')
'ACGTG'
>>> ['dog', 'cat', 'bird'].rstrip()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'rstrip'
```

What functions go with my object?

How do you find out what functions work with an object? There's a handy function `dir()`. As an example what functions can you call on our string `'ACGTGA'`?

```
>>> dir('ACGTGA')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
   '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
   '__getitem__', '__getnewargs__', '__gt__', '__hash__',
   '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
   '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
   '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
   '__rmul__', '__setattr__', '__sizeof__', '__str__',
   '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
   'encode', 'endswith', 'expandtabs', 'find', 'format',
   'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
   'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
   'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
   'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
   'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
   'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
   'zfill']
```

You can call `dir()` on any object.

Strings

- A string is a series of characters starting and ending with a quotation mark.
- Strings are an example of a Python Sequence. A sequence is defined as a positionally ordered set. This means each element in the set has a position, starting with zero, i.e. 0,1,2,3 and so on until you get to the end of the string. If this is confusing, think about a string as being made up of individual characters: character 0, character 1, character 2 and so on.

Quotation Marks

- Single (')
- Double (“)
- Triple (‘” or “”“)

Notes about quotes:

- Single and double quotes are the same.
- A variable will not be replaced with its value (sometimes called 'interpolation') if placed inside of quotes.
- Triple quotes are used before and after a string that spans multiple lines.

Use of quotation examples:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences.
"""
```

Strings and the `print()` function

We saw examples of `print()` earlier. Lets talk about it a bit more.

`print()` is a function that takes one or more comma-separated arguments.

Let's use the `print()` function to print a string.

```
>>>print("ATG")
ATG
```

> We get ATG printed to the screen like we expect.

Let's assign a string to a variable and print the variable.

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
```

> We get ATG printed to the screen

What happens if we put the variable in quotes?

```
>>>dna = 'ATG'
ATG
>>> print("dna")
dna
```

The literal value of 'dna' is printed to the screen. The variable called 'dna' is not interpolated when it is inside of quotes.

Let's see what happens when we give `print()` two literal strings as arguments.

```
>>> print("ATG","GGTCTAC")
ATG GGTCTAC
```

We get the two literal strings printed to the screen separated by a space

What if you do not want your strings separated by a space? Use the concatenation operator to concatenate the two strings before or within the `print()` function.

```
>>> print("ATG"+"GGTCTAC")
ATGGGTCTAC
>>> combined_string = "ATG"+"GGTCTAC"
ATGGGTCTAC
>>> print(combined_string)
ATGGGTCTAC
```

We get the two strings printed to the screen without being separated by a space.
You can also use this

```
>>> print('ATG','GGTCTAC',sep='')
ATGGGTCTAC
```

Now, lets print a variable and a literal string.

```
>>>dna = 'ATG'
ATG
>>> print(dna,'GGTCTAC')
ATG GGTCTAC
```

We get the value of the variable and the literal string printed to the screen separated by a space

How would we print the two without a space?

```
>>>dna = 'ATG'
ATG
>>> print(dna + 'GGTCTAC')
ATGGGTCTAC
```

Something to think about: Values of variable are variable. Or in other words, they are mutable, changeable.

```
>>>dna = 'ATG'
ATG
>>> print(dna)
ATG
>>>dna = 'TTT'
TTT
>>> print(dna)
TTT
```

The new value of the variable 'dna' is printed to the screen when 'dna' is an agrument for the `print()` function.

Errors and Printing

Let's look at the typical errors you will encouter when you use the `print()` function.

What will happen if you forget to close your quotes?

```
>>> print("GGTCTAC)
  File "<stdin>", line 1
    print("GGTCTAC)
                  ^
SyntaxError: EOL while scanning string literal
```

We get a 'SyntaxError' if the closing quote is not used

What will happen if you forget to enclose your literal string in quotes in a print statement?

```
print(GGTCTAC)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'GGTCTAC' is not defined
```

We get a 'NameError' when the literal string is not enclosed in quotes.

```
>>> print "boo"
  File "<stdin>", line 1
    print "boo"
              ^
SyntaxError: Missing parentheses in call to 'print'
```

In python2, the command was `print`, but this changed in python3 to `print()`, so don't forget the parentheses!

Special/Escape Characters

How would you include a new line, carrage return, or tab in your string?

Escape Character	Description
\n	New line
\r	Carrage Return
\t	Tab

Let's include some escape charcters in our strings and print() functions.

```
>>> string_with_newline = 'this sting has a new line\nthis is the
    second line'
>>> print(string_with_newline)
this sting has a new line
this is the second line
```

We printed a new line to the screen

Generally, you don't have to worry about adding whitespace between arguments because `print()` adds space between arguments and a new line at the end for you. You can change these with sep= and end= `print('one line', 'second line' , 'third line', sep='\n', end = '')`

A neater way to do this is to express a multi-line string enclosed in triple quotes (“”“).

```
>>> print("""this sting has a new line
... this is the second line""")
this sting has a new line
this is the second line
```

Let's print a tab character ().

```
>>> line = "value1\tvalue2\tvalue3"
>>> print(line)
value1  value2  value3
```

> We get the three words separated by tab characters. A common format for data is tab separated.

You can add a backslash before any chacter to force it to be printed as a literal. This is called 'escaping'. This is only really useful for printing literal quotes ' and "

```
>>> print("this is a \'word\'")
this is a 'word'
>>> print("this is a 'word'")
this is a 'word'
```

> In both cases the single quote is printed to the screen as a quote.

If you want every character in your string to remain exactly as it is, declare your string a raw string literal with 'r' before the first quote. This looks ugly, but it works.

```
>>> line = r"value1\tvalue2\tvalue3"
>>> print(line)
value1\tvalue2\tvalue3
```

> Our escape characters '' remain as we typed them, they are not converted to actual tab characters.

Concatenation

To concatenate strings use the concatenation operator '+'

```
>>> promoter= 'TATAAA'
>>> upstream = 'TAGCTA'
>>> downstream = 'ATCATAAT'
>>> dna = upstream + promoter + downstream
>>> print(dna)
TAGCTATATAAAATCATAAT
```

The concatenation operator can be used to combine strings. The newly combined string can be stored in a variable. What happens if you use + with numbers (these are integers or ints)?

```
>>> 4+3
7
```

For strings, + concatenates; for integers, + adds.

You need to convert the numbers to strings before you can concatenate them

```
>>> str(4) + str(3)
'43'
```

Determine the length of a string

Use the `len()` function to calculate the length of a string. This function takes a sequence as an argument and returns an int

```
>>> print(dna)
TAGCTATATAAAATCATAAT
>>> len(dna)
20
```

The length of the string, including spaces, is calculated and returned.

The value`len()` returns can be stored in a variable.

```
>>> dna_length = len(dna)
>>> print(dna_length)
20
```

You can mix strings and ints in `print()`, but not in concatenation.

```
>>> print("The lenth of the DNA sequence:" , dna , "is" , dna_length)
The lenth of the DNA sequence: TAGCTATATAAAATCATAAT is 20
```

Changing String Case

Changing the case of a string is a bit different that you might first expect. For example, to lowercase a string we need to use a method. A method is a function that is specific to a object. When we assign a string to a variable we are creating an instance of a string object. This object has a series of methods that will work on the data that is stored in the object. `lower()` function is one of these object methods.

Let's do create a new string object.

```
dna = "ATGCTTG"
```

> Look familiar? It should. Creating a string object is what we have been doing all along.

Now that we have a string object we can use string methods. The way you use a method is to append the method with a '.' to the variable name.

```
>>> dna = "ATGCTTG"
>>> dna.lower()
'atgcttg'
```

> the lower() method returns the contents stored in the 'dna' variable in lowercase.

The contents of the 'dna' variable have not been changed. If you want to reuse the returned lowercased value, store it in a new variable.

```
>>> print(dna)
ATGCTTG
>>> dna_lowercase = dna.lower()
>>> print(dna)
ATGCTTG
>>> print(dna_lowercase)
atgcttg
```

The string method can be nested inside of other functions.

```
>>> dna = "ATGCTTG"
>>> print(dna.lower())
atgcttg
```

The value of 'dna' is lowercased and returned. The `print()` function takes the returned value from the `lower()` method and prints it.

If you try to use a string method on a object that is not a string you will get an error.

```
>>> nt_count = 6
>>> dna_lc = nt_count.lower()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'lower'
```

You get an AttributeError when you use a method on the an incorrect object type. We are told that the int object (an int is returned by `len()`) does not have a function called lower.

Now let's uppercase a string.

```
>>> dna = 'attgct'
>>> dna.upper()
'ATTGCT'
>>> print(dna)
attgct
```

The contents of the variable 'dna', 'attgct' was returned in upper case. The actual contents of 'dna' were not altered.

Find and Count

`count(str)` returns the number of exact matches of `str` it found (as an int)

```
>>> dna = 'ATGCTGCATT'
>>> dna.count('T')
4
```

The number of times 'T' is found is returned. The string stored in 'dna' is not altered.

Find and Replace

`replace(str1,str2)` returns a new string with all matches of `str1` in a string replaced with `str2`.

```
>>> dna = 'ATGCTGCATT'
>>> dna.replace('T','U')
'AUGCUGCAUU'
>>> print(dna)
ATGCTGCATT
>>> rna = dna.replace('T','U')
>>> print(rna)
AUGCUGCAUU
```

All occurrences of T are replaced by U. The new string is returned. The orginal string has not actually been altered. If you want to reuse the new string, store it in a variable.

Extracting a Substring, or Slicing

Parts of a string can be located based on position and returned. This is because a string is a sequence. Coordinates start at 0. You add the coordinate in square brackets after the string's name.

This string 'ATTAAAGGGCCC' is made up of the following sequence of characters, and positions (starting at zero).

Position/Index	Character
0	A
1	T
2	T
3	A
4	A
5	A
6	G
7	G
8	G
9	C
10	C
11	C

Let's return the 4th, 5th, and 6th nucleotides. To do this, we need to count like a computer and start our string at 0 and return the 3rd, 4th, and 5th characters. This will be everything from 3 to 6.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[3:6]
>>> print(sub_dna)
AAA
```

The chacters with the positional index of 3, 4, 5 are returned. Or in other words, every chacater starting at index 3 and up to but not including, the index of 6 are returned.

Let's return the first 6 characters.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[0:6]
>>> print(sub_dna)
ATTAAA
```

Every character starting at index 0 and up to but not including index 6 are returned. This is the same as dna[:6]

Let's return every character from index 6 to the end of the string.

```
>>> dna = 'ATTAAAGGGCCC'
>>> sub_dna = dna[6:]
>>> print(sub_dna)
GGGCCC
```

When the second argument is left blank, every character from index 6 and greater is returned.

Let's return the last 3 characters.

```
>>> sub_dna = dna[-3:]
>>> print(sub_dna)
CCC
```

When the second argument is left blank and the first argument is negative (-X), X characters from the end of the string are returned.

Locate and Report

The positional index of an exact string in a larger string can be found and returned with the string method `find` . A exact string is given as an argument and the index of its first occurrence is returned. -1 is returned if it is not found.

```
>>> dna = 'ATTAAAGGGCCC'
>>> dna.find('T')
1
>>> dna.find('N')
-1
```

The substring 'T' is found for the first time at index 1 in the string 'dna' so 1 is returned. The substring 'N' is not found, so -1 is returned.

Other String Methods

Since these are methods, be sure to use in this format string.method().

function	Description
s.strip()	returns a string with the whitespace removed from the start and end
s.isalpha()	tests if all the characters of the string are alphabetic characters. Returns True or False.
s.isdigit()	tests if all the characters of the string are numeric characters. Returns True or False.
s.startswith('other_string')	tests if the string starts with the string provided as an argument. Returns True or False.
s.endswith('other_string')	tests if the string ends with the string provided as an argument. Returns True or False.
s.split('delim')	splits the string on the given exact delimiter. Returns a list of substrings. If no argument is supplied, the string will be split on whitespace.
s.join(list)	opposite of split(). The elements of a list will be concatenated together using the string stored in 's' as a delimiter.

String Formatting

Strings can be formated using the `format()` function. Pretty intuitive! For example, if you want to include literal stings and variables in your print statement and do not want to concatenate or use multiple arguments in the `print()` function you can use string formatting.

```
>>> string = "This sequence: {} is {} nucleotides long and is found
    in {}."
>>> string.format(dna,dna_len,gene_name)
'This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and
    is found in Brca1.'
>>> print(string)
This sequence: {} is {} nucleotides long and is found in {}.
>>> new_string = string.format(dna,dna_len,gene_name)
>>> print(new_string)
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is
    found in Brca1.
```

We put together the three variables and literal strings into a single string using the function `format()`. The original string is not altered, a new string is returned that incorporates the arguments. You can save the returned value in a new variable. Each `{}` is a placeholder for the strings that need to be inserted.

> Something nice about `format()` is that you can print int and string variable types without converting first.

You can also directly call the format function on a string inside a print function. Here are two examples

```
>>> string = "This sequence: {} is {} nucleotides long and is found
    in {}."
>>> print(string.format(dna,dna_len,gene_name))
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is
    found in Brca1.
```

Or you can create a string on the fly and use the `format()` function.

```
>>> print( "This sequence: {} is {} nucleotides long and is found in
    {}.".format(dna,dna_len,gene_name))
This sequence: TGAACATCTAAAAGATGAAGTTT is 23 nucleotides long and is
    found in Brca1.
```

> There is no need to store the string in a variable.

The `format()` mini-language

So far, we have just used `{}` to show where to insert the value of a variable in a string. You can add special characters inside the `{}` to change the way the variable is formatted when it's inserted into the string.

> You can number these, not necessarily in order.

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
```

To change the spacing of strings and the way numbers are formatted, you add `:` and other special characters like this `{:>5}` to right-justify a string in a five-character field.

Lets right justify some numbers.

```
>>> print( "{:>5}".format(2) )
    2
>>> print( "{:>5}".format(20) )
   20
>>> print( "{:>5}".format(200) )
  200
```

> The arguments: 2, 20, 200 have all been right justified in a field that is 5 characters wide by using '{:>5}'

How about padding with zeroes? This means the five-character field will be filled as needed with zeroes to the left of any numbers you want to display

```
>>> print( "{:>05}".format(2) )
00002
>>> print( "{:>05}".format(20) )
00020
```

> Now all of the arguments: 2 ,20 are right justified to a witdh of 5 and any empty space is filled with a zero.

Use a `<` to indicate left-justification.

```
>>> print( "{:<5} genes".format(2) )
2     genes
>>> print( "{:<5} genes".format(20) )
20    genes
>>> print( "{:<5} genes".format(200) )
200   genes
```

> The arguments: 2, 20, 200 have all been left justified by a width of 5 columns by using '{:<5}'

If you want to pad with a non-zero character, you can add this too. Between : and the symbol for the justification. Let's try padding with _, an underscore.

```
>>> print( "{:_<5} next".format(2) )
2____ next
>>> print( "{:_<5} next".format(20) )
20___ next
>>> print( "{:_<5} next".format(200) )
200__ next
```

The arguments: 2, 20, 200 have all been left justified in a field that is 5 characters wide and any empty space is filled with a _

Center aligning is done with ^

```
>>> print( "{:_^10}".format(2) )
____2_____
>>> print( "{:_^10}".format(20) )
____20____
>>> print( "{:_^10}".format(200) )
___200____
```

Text can be center aligned by using ':^10'. 10 of course is your column width. The '^' indicates center justification. In our example an underscore is used to illustrate the empty spaces.

Summary of special formatting symbols so far

Here are some of the ALIGNMENT options:

Option	Meaning
'<'	Forces the field to be left-aligned within the available space (this is the default for most objects).
'>'	Forces the field to be right-aligned within the available space (this is the default for numbers).
'='	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form '+000000120'. This alignment option is only valid for numeric types.
'^'	Forces the field to be centered within the available space.

Here's an example

`{ : x < 10 s}`

fill with `x`

left justify `<`
`10` a field of ten characters `s` a string

Common Types

type	description
b	convert to binary
d	decimal integer
e	exponent, default precision is 6, uses `e`
E	exponent, uses `E`
f	floating point, default precision 6 (also F)
g	general number, float for values close to 0, exponent for others; also G
s	string, default type
x	convert to hexidecimal, also X
%	converts to % by multiplying by 100

What's the point?

So much can be done with the `format()` function. Here is one last example, but not the last functionality of this function. Let truncate a long floating point number. The default is 6 decimal places. Note that the function rounds to the nearest decimal place.

```
1 '{:f}'.format(3.141592653589793)
2 '3.141593'
3 >>> '{:.4f}'.format(3.141592653589793)
4 '3.1416'
```

Lists and Tuples

Lists

Lists are valuable data types that can store a collection of data in a single variable.

- Lists are used to store an ordered, *indexed* collection of data.
- Values are separated by commas
- Values are enclosed in square brackets '[]'
- Lists can grown and shrink
- Values are mutatable

Accessing Values in Lists

To retrieve a single value in a list use the value's index in this format list[index]. This will return the value at the specified index.

Here is our list:

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
```

There are 3 values with the indices of 0, 1, 2

Index	Value
0	atg
1	aaa
2	agg

Let's access the 0th value.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons[0]
'atg'
```

The the value of the 0th index is returned by using the syntax list[index]

The value cana be saved for later use by storing in a variable.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codon_0 = codons[0]
>>> print(codon_0)
atg
```

Each value can be saved in a new variable to use later.

The values can be retrieved an used directly.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[0])
atg
>>> print(codons[1])
aaa
>>> print(codons[2])
agg
```

The 3 values are independently accesses and immediately printed. They are not stored in a variable.

If you want to access the values in reverse, use negative indices.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> print(codons[-1])
agg
>>> print(codons[-2])
aaa
```

Using a negative index will return the values from the end of the list. For example, -1 is the index of the last value 'agg'. This value also has an index of 2.

Changing Values in a List

Indivudual values can be changed using the value's index and the assignment operator.

```
>>> print(codons)
['atg', 'aaa', 'agg']
>>> codons[2] = 'cgc'
>>> print(codons)
['atg', 'aaa', 'cgc']
```

codon[2] used to contain 'agg'. We reassigned condon[2] to contain the new value 'cgc'

What about trying to assign a value to an index that does not exist?

```
>>> codons[5] = 'aac'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

codon[5] does not exist, and when we try to assign a value to this index we get an IndexError.

Exracting a Subset of a List, or Slicing

This works in exactly the same way with lists as it does with strings. This is because both are Sequences, or ordered collections of data with positional information.

Index	Value
0	atg
1	aaa
2	agg
3	aac

Index	Value
4	cgc
5	acg

```
>>> codons = [ 'atg' , 'aaa' , 'agg' , 'aac' , 'cgc' , 'acg']
>>> print (codons[1:3])
['aaa', 'agg']
>>> print (codons[3:])
['aac', 'cgc', 'acg']
>>> print (codons[:3])
['atg', 'aaa', 'agg']
>>> print (codons[0:3])
['atg', 'aaa', 'agg']
```

codons[1:3] returns every value starting with the value of codons[1] up to but not including codons[3] codons[3:] returns every value starting with the value of codons[3] and every value after. codons[:3] returns every value up to but not including codons[3] codons[0:3] is the same as codons[:3]

List Operators

Operator	Description	Example
+	Concatenation	[10, 20, 30] + [40, 50, 60] returns [10, 20, 30, 40, 50, 60]
*	Repetition	[‘atg’] * 4 returns [‘atg’,‘atg’,‘atg’,‘atg’]
in	Membership	20 in [10, 20, 30] returns True

List Functions

Functions	Description	Example
len(list)	returns the length or the number of values in list	len([1,2,3]) returns 3
max(list)	returns the value with the largest ascii value	max([‘a’,‘A’,‘z’]) returns ‘z’
min(list)	returns the value with the smallest ascii value	min([‘a’,‘A’,‘z’]) returns ‘A’
list(seq)	converts a tuple into a list	list((‘a’,‘A’,‘z’)) returns [‘a’, ‘A’, ‘z’]
sorted(list, key=None, reverse=False)	returns a sorted list based on the key provided	sorted([‘a’,‘A’,‘z’]) returns [‘A’, ‘a’, ‘z’] sorted([‘a’,‘A’,‘z’],key=str.lower) returns [‘a’, ‘A’, ‘z’]

List Methods

Remember methods are apart of the object and are used in the following format list.method().

For these examples use: list = [1,2,3] and codons = [‘atg’ , ‘aaa’ , ‘agg’]

Method	Description	Example
list.append(obj)	appends an object to the end of a list	list.append(9) ; print(list) ; returns [1,2,3,9]
list.count(obj)	counts the occurance of an object in a list	list.count(2) returns 1
list.index(obj)	returns the lowest index where the given object is found	list.index(2) returns 1
list.pop()	removes and returns the last value in the list. The list is now 1 value shorter	list.pop() returns 3
list.insert(index, obj)	inserts a value at the given index	list.insert(0,100) ; print(list) returns [100, 1, 2, 3]
list.extend(new_list)	adds the provided list to the end of list	list.extend([‘a’, ‘z’]) ; print(list) returns [1, 2, 3, ‘a’, ‘z’]
list.pop(index)	removes and returns the value of the index argument. The list is now 1 value shorter	list.pop(0) returns 1
list.remove(obj)	finds the lowest index of the given object and removes the value. The list is now 1 value shorter	codons.remove(‘aaa’) ; print(codons) returns [‘atg’ , ‘agg’]
list.reverse()	reverses the order of the list	list.reverse() ; print(list) returns [3,2,1]
list.copy()	Returns a shallow copy of list. Shallow vs Deep only matters in multidementional datastructures.	
list.sort([func])	sorts a list using the provided function. Does not return a list. The list has been changed. Advanced list sort will be covered once writting your own functions has been disccussed.	codons.sort() ; print(codons) returns [‘aaa’, ‘agg’, ‘atg’]

Be careful how you make a copy of your list

```
>>> list=['a', 'one', 'two']
>>> l2=list
>>> l2.append('1')
>>> print(list)
['a', 'one', 'two', '1']
>>> print(l2)
['a', 'one', 'two', '1']
```

> Not what you expected?! Boht lists have changed because we only copied a pointer to the original list when we wrote `l2=list`.

Let’s copy the list using the `copy()` method.

```
>>> list=['a', 'one', 'two']
>>> l2=list.copy()
>>> l2.append('1')
>>> print(list)
['a', 'one', 'two']
```

There we go, we get what we expect this time!

Building a List one Value at a Time

Now that you have seen the `append()` function we can go over how to build a list one value at a time.

```
>>> words = []
>>> print(words)
[]
>>> words.append('one')
>>> words.append('two')
>>> print(words)
['one', 'two']
```

We start with a an empty list called 'words'. We use `append()` to add the value 'one' then to add the value 'two'. We end up with a list with two values.

Link to Python 3 Problem Set

... "' > What caused this code to never be true? > The statement that increments the count is missing. To stop the code from forever printing use Cntl+C.

While/Else

An Else statment can be used with a while statement. It behaves in the same way as with an If statement. When the while statement is false, the else block is excuted ONE TIME.

```
#!/usr/bin/python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
else:
  print("count:", count, "is now not less than 5")
print("Done")
```

Output:

```
$ python while_else.py
count: 0
count: 1
count: 2
count: 3
count: 4
count: 5 is now not less than 5
Done
```

> The while loop was executed five times like before. Now when count is equal to 5 and therefore not less than 5, the else block is executed. Finally the lines of code outside the while/else are executed.

For Loops

A for loop is a loop that executes the for block of code for every iteration of a sequence. Remember a sequence is an ordered collection of data.

For Loop Syntax

```
for iterating_variable in sequence:
  statement(s)
```

An example of a sequence is a list. Let's use a for loop list of words.

Code:

```
#!/usr/bin/python3

words = ['zero','one','two','three','four']
for word in words:
  print(word)
```

Output:

```
python3 list_words.py
zero
one
two
three
four
```

This is an example of iterating over a string. Remember a string is a sequence like a list. It is data where position is important. Look back at "Extracting a Substring, or Slicing" in the Strings section to see other ways of how strings can be treated like lists.

Code:

```
#!/usr/bin/python3

dna =
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
for nt in dna:
  print(nt)
```

Output:

```
$ python3 for_string.py
G
T
A
C
C
T
T
...
...
```

This is a easy way to access each character in a string. It is especially nice for DNA sequences.

Another example of iterating over a list of variables, this time numbers.

Code:

```
#!/usr/bin/python3

numbers = [0,1,2,3,4]
for num in numbers:
  print(num)
```

Output:

```
$ python3 list_numbers.py
0
1
2
3
4
```

Python has a function called `range()` that will return numbers that can be converted to a list.

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

This can be used in conjunction with a for loop to iterate over a range of numbers Code:

```
#!/usr/bin/python3

for num in range(5):
  print(num)
```

Output:

```
$ python list_range.py
0
1
2
3
4
```

As you can see this is the same output as using the list numbers = [0, 1, 2, 3, 4] And this has the same functionality as a while loop with a condition of count = 0 ; count < 5.

For/Else

An else statement can be used with a for loop as well. The else block of code will be executed when the for loop exits normally.

> Can we just skip this. it seems pretty stupid

Loop Control

Loops control statements allow for altering the normal flow of execution.

Control Statement	Description
break	A loop is terminated when a break statement is executed. All the lines of code after the break, but within the loop block are not executed. No more iteration of the loop are preformed
continue	A single iteration of a loop is terminated when a continue statement is executed. The next iteration will proceed normally.

Loop Control: Break

Code:

```
#!/usr/bin/python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    break
print("Done")
```

Output:

```
$ python break.py
count: 0
count: 1
count: 2
Done
```

> when the count is equal to 3, all iterations of the while loop are terminated

Loop Control: Continue

Code:

```
#!/usr/bin/python3

count = 0
while count < 5:
  print("count:" , count)
  count+=1
  if count == 3:
    continue
  print("line after our continue")
print("Done")
```

Output:

```
$ python continue.py
count: 0
line after our continue
count: 1
line after our continue
count: 2
count: 3
line after our continue
count: 4
line after our continue
Done
```

When the count is equal to 3 the continue is executed. This causes all the lines within the loop block to be skipped. “line after our continue” is not printed when count is equal to 3. The next loop is executed normally.

Iterators

An iterable is any data type that is iterable, or can be used in iteration. An iterable can be made into an iterator with the `iter()` function. This means you can use the `next()` function.

```
>>> codons = [ 'atg' , 'aaa' , 'agg' ]
>>> codons_iterator=iter(codons)
>>> next(codons_iterator)
'atg'
>>> next(codons_iterator)
'aaa'
>>> next(codons_iterator)
'agg'
>>> next(codons_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

An iterator allows you to get the next element in the iterator until there are no more elements. If you want to go through each element again, you will need to redefine the iterator.

List Comprehension

List comprehension is a way to make a list without typing out each element. There are many many ways to use list comprehension to generate lists. Some are quite complex, yet useful.

Here is an simple example:

```
dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
lengths = [len(dna) for dna in dna_list]
```

This will produce this list [4, 8, 3, 8]

Using conditions:

```
dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
lengths = [len(dna) for dna in dna_list if dna.startswith('A')]
```

This generates the following list: [8, 3, 8]

Using conditions:

```
dna_list = ['TAGC', 'ACGTATGC', 'ATG', 'ACGGCTAG']
lengths = [len(dna) for dna in dna_list if dna.startswith('A')]
```

This generates the following list: [8, 3, 8]

Here is an example of using mathatical operators to generate a list:

```
l = [2 ** x for x in range(100000)]
```

This creates a list of list of the first one hundred thousand powers of two

Dictionaries

Dictionaries are another iterable, like a string and list. Unlike strings and lists, dictionaries are not a sequence, or in other words, the position is not important.

Dictionaries are a collection of key/value pairs. In python, each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: `{}`

Each key in a dictionary is unique, while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Data that is appropriate for dictionaries are two pieces of information that naturally go together, like gene name and sequence.

Key	Value
TP53	GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAG
BRCA1	GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCG

Creating a Dictionary

```
genes = { 'TP53' :
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
    , 'BRCA1' :
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
    }
```

Breaking up the key/value pairs over multiple lines make them easier to read.

```
genes = {
            'TP53' :
               'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGT
               ,
            'BRCA1' :
               'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAA
         }
```

Accessing Values in Dictionaries

To retrieve a single value in a dictionary use the value's key in this format `dict[key]`. This will return the value at the specified key.

```
>>> genes = { 'TP53' :
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
    , 'BRCA1' :
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
    }
>>>
>>>
>>> genes['TP53']
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

> The sequence of the gene TP53 is stored as a value of the key 'TP53'. We can assess the sequence by using the key in this format dict[key]

The value can be accessed and passed directly to a function or stored in a variable.

```
>>> print(genes['TP53'])
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
>>> seq = genes['TP53'];
>>> print(seq)
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
```

Changing Values in a Dictionary

Individual values can be changed via the key and the assignment operator.

```
>>> genes = { 'TP53' :
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
    , 'BRCA1' :
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
    }
>>> print(genes)
{'BRCA1':
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA',
    'TP53':
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
>>> genes['TP53'] = 'atg'
>>> print(genes)
{'BRCA1':
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA',
    'TP53': 'atg'}
```

The contents of the dictionary have changed.

Other assignment operators can also be used to change a value of a dictionary key.

```
>>> genes = { 'TP53' :
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
    , 'BRCA1' :
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
    }
>>> genes['TP53'] +=
    'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'
>>> print(genes)
{'BRCA1':
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA',
    'TP53':
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTCTAGAGCCACC
```

Here we have used the '+=' concatenation assignemt operator. This is equivalent to genes['TP53'] = genes['TP53'] + 'TAGAGCCACCGTCCAGGGAGCAGGTAGCTGCTGGGCTCCGGGGACACTTTGCGTTCGGGCTGGGAGCGTG'.

Building a Dictionary one Key/Value at a Time

Building a dictionary one key/value at a time is akin to what we just saw when we change a key's value. Normally you won't do this. We'll talk about ways to build a dictionary from a file in a later lecture.

```
>>> genes = {}
>>> print(genes)
{}
>>> genes['Brca1'] =
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA'
>>> genes['TP53'] =
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'
>>> print(genes)
{'Brca1':
    'GTACCTTGATTTCGTATTCTGAGAGGCTGCTGCTTAGCGGTAGCCCCTTGGTTTCCGTGGCAACGGAAAA',
    'TP53':
    'GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC'}
```

We start by creating an empty dictionary. Then we add each key/value pair using the same syntax as when we change a value.
dict[key] = new_value

Checking That Dictionary Keys Exist

Python generates an error (NameError) if you try to access a key that does not exist.

```
>>> print(genes['HDAC'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'HDAC' is not defined
```

Operator	Descsription
in	key in dict returns True if the key exists in the dictionary
not in	key not in dict returns True if the key does not exist in the dictionary

Because python generates a NameError if you try to use a key that doesn't exist in the dictionary, you probably need to check whether a key exists before trying to use it. The best way to check whether a key exists is to use `in`

```
>>> gene = 'TP53'
>>> if gene in genes: print('found')
...
found
>>>
>>> if gene in genes:
...     print(genes[gene])
...
GATGGGATTGGGGTTTTCCCCTCCCATGTGCTCAAGACTGGCGCTAAAAGTTTTGAGCTTCTCAAAAGTC
>>>
```

Sorting Dictionary Keys

If you want to print the contents of a dictionary, you probably want to sort the keys then iterate over the keys with a for loop. Why do you want to sort the keys?

```
for key in sorted(genes):
  print(key, '=>' , genes[key])
```

This will print the same order of keys every time you run your script. This is good for finding bugs and means you don't have to check whether the key exists explicitly because `in sorted(genes)` returns a list of all the keys. It also sorts it.

Dictionary Functions

Function	Description
len(dict)	returns the total number of key/value pairs
str(dict)	returns a string representation of the dictionary
type(variable)	Returns the type or class of the variable passed to the function. If the variable is dictionary, then it would return a dictionary type.

Dictionary Methods

Method	Description
dict.clear()	Removes all elements of dictionary dict
dict.copy()	Returns a shallow copy of dictionary dict. Shallow vs Deep only matters in multidementional datastructures.
dict.fromkeys(seq,value)	Create a new dictionary with keys from seq (python sequence type) and values set to value.
dict.items()	Returns a list of (key, value) tuple pairs
dict.keys()	Returns list of keys
dict.setdefault(key, default = None)	Similar to get(), but will set dict[key] = default if key is not already in dict
dict.update(dict2)	Adds dictionary dict2's key-values pairs to dict

Method	Description
dict.values()	Returns list of dictionary dict's values

Sets

A set is another Python data type. It is essentially a dictionary with keys but no values.

- A set is unordered
- A set is a collection of data with no duplicate elements.
- Common uses include looking for differences and eliminating duplicates in data sets.

Curly braces or the `set()` function can be used to create sets.

> Note: to create an empty set you have to use `set()`, not `{}` the latter creates an empty dictionary.

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)
{'orange', 'banana', 'pear', 'apple'}
```

> Look, duplicates have been removed

Test to see if an value is in the set

```
>>> 'orange' in basket
True
>>> 'crabgrass' in basket
False
```

> The in operator works the same with sets as it does with lists and dictionaries

Union, intersection, difference and symmetric difference can be done with sets

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
```

> Sets contain unique elements, therefore, even if duplicate elements are provided they will be removed.

Set Operators

Difference

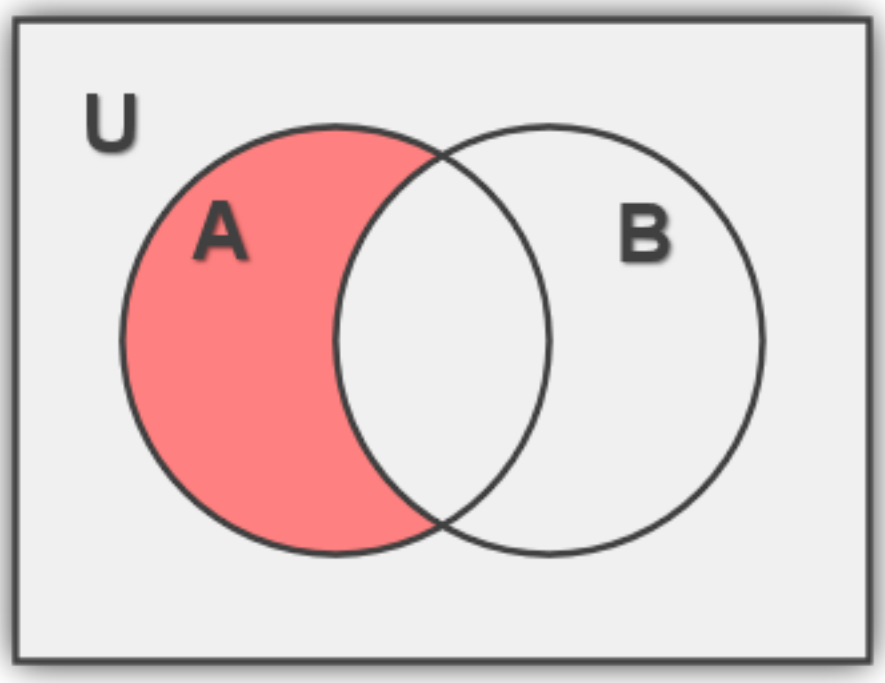

Figure 2: Set Difference

```
>>> a - b
{'r', 'd', 'b'}
```

This returns the letters that are in a but not in b

Union

```
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
```

This returns letters that are in a or b both

Intersection

```
>>> a & b
{'a', 'c'}
```

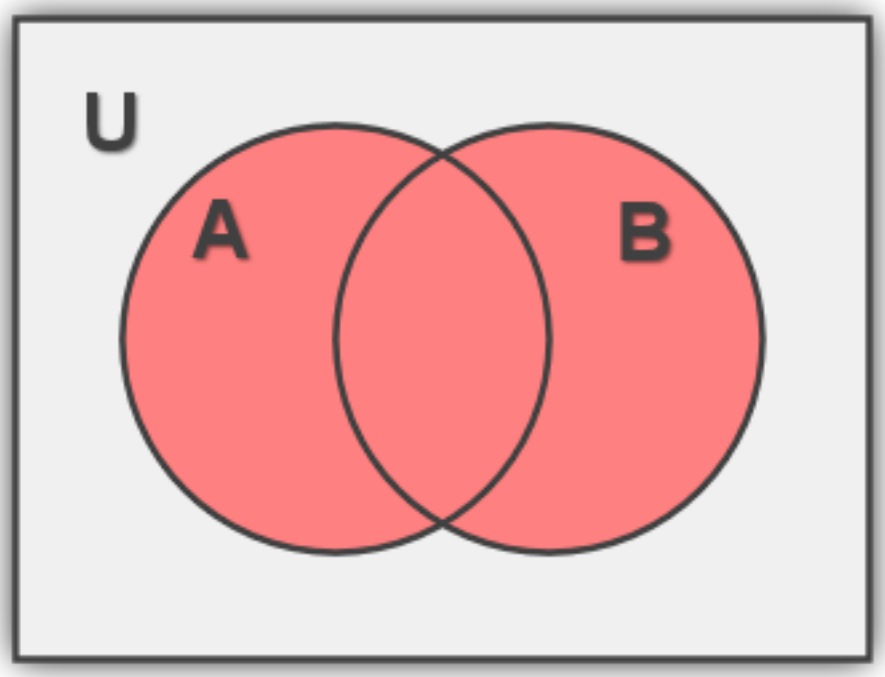

Figure 3: Set Union

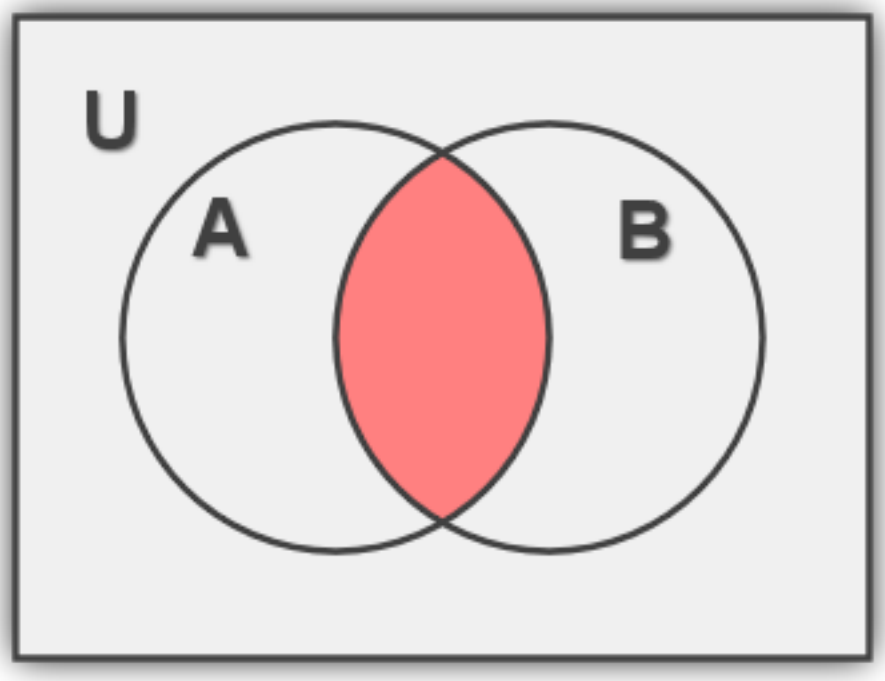

Figure 4: Set Intersection

This returns letters that are in both a and b

Symmetric Difference

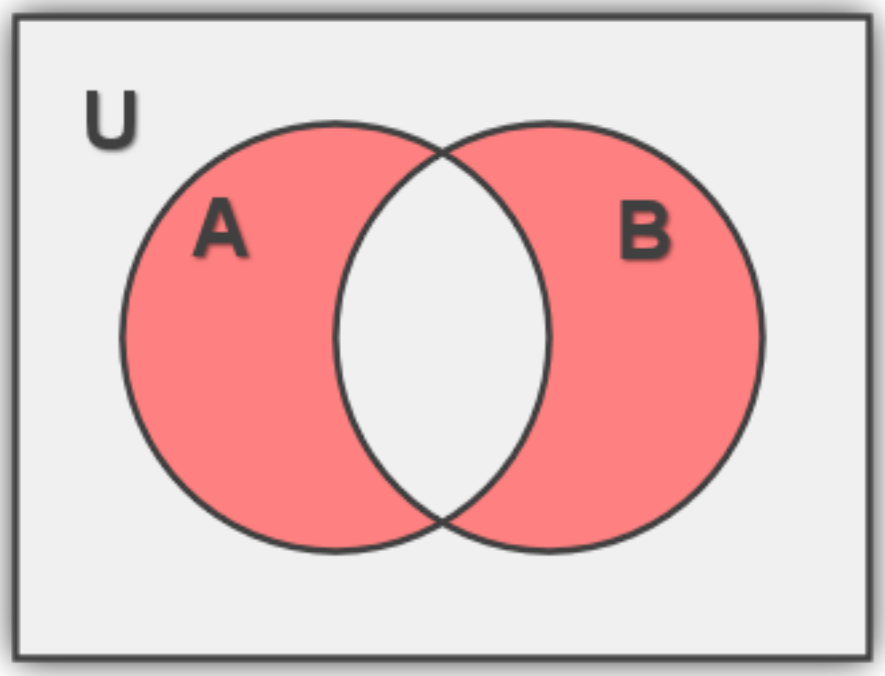

Figure 5: Set Symmetric Difference

```
>>> a ^ b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

This returns the letters that are in a or b but not in both (also known as exclusive or)

Set Functions

Function	Description
all()	returns True if all elements of the set are true (or if the set is empty).
any()	returns True if any element of the set is true. If the set is empty, return False.
enumerate()	return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	return the number of items in the set.
max()	return the largest item in the set.
min()	return the smallest item in the set.

Function	Description
sorted()	returns a new sorted list from elements in the set (does not alter the original set).
sum()	retrune the sum of all elements in the set.

Set Methods

Method	Description
set.add(new)	adds a new element
set.clear()	remove all elements
set.copy()	returns a shallow copy of a set
set.difference(set2)	returns the difference of set and set2
set.difference_update(set2)	removes all elements of another set from this set
set.discard(element)	removes an element from set if it is found in set. (Do nothing if the element is not in set)
set.intersection(sets)	return the intersection of set and the other provided sets
set.intersection_update(sets)	updates set with the intersection of set and the other provided sets
set.isdisjoint(set2)	returns True if set and set2 have no intersection
set.issubset(set2)	returns True if set2 contains set
set.issuperset(set2)	returns True if set contains set2
set.pop()	removes and returns an arbitary element of set.
set.remove(element)	removes element from a set.
set.symmetric_difference(set2)	returns the symmetric difference of set and set2
set.symmetric_difference_update(set2)	updates set with the symmetric difference of set and set2
set.union(sets)	returns the union of set and the other provided sets
set.update(set2)	update set with the union of set and set2

Link to Python 4 Problem Set

Link to Python 5 Problem Set

Found an EcoRI site! >>> "`> Since we can search for control characters like a tab (\t)it is good to get in the habit of using the raw string function >r`‘ > when defining patterns.

> Here we used the `search()` function with two arguments, 1) our pattern and 2) the string we want to search.

Let's find out what is returned by the `search()` function.

```
>>> dna =
    'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCC
>>> found=re.search(r"GAATTC",dna)
>>> print(found)
<_sre.SRE_Match object; span=(70, 76), match='GAATTC'>
```

> Information about the first match is returned

How about a non-exact match. Let's seach for a methylation site that has to match the following critera: - G or A - followed by C - followed by one of anything or nothing - followed by a G

This could match any of these:
GCAG
GCTG
GCGG
GCCG
GCG
ACAG
ACTG
ACGG
ACCG
ACG

We could test for each of these, or use regular expressions. This is exactly what regular expressions can do for us.

```
>>> dna =
    'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCC
>>> found=re.search(r"[GA]C.?G",dna)
>>> print(found)
<_sre.SRE_Match object; span=(7, 10), match='ACG'>
```

> Here you can see in the returned infomratin that ACG starts at string postion 7 (nt 8). The first postion not in the match is at string postion 10 (nt 11).

What about other potential matches in our DNA string? We can use `findall()` function to find all matches.

```
>>> dna =
    'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCC
>>> found=re.findall(r"[GA]C.?G",dna)
>>> print(found)
['ACG', 'GCTG', 'ACTG', 'ACCG', 'ACAG', 'ACCG', 'ACAG']
```

`findall()` returns a list of all the pieces of the string that match the regex.

A quick count of all the matching sites can be done by counting the length of the returned list.

```
>>> len (re.findall(r"[GA]C.?G",dna))
7
```

There are 7 methylation sites. Here we have another example of nesting. We call the `findall()` function, searching for all the matches of a methylation site. This function returns a list, the list is past to the `len()` function, which in turn returns the number of elements in the list.

Let's talk a bit more about all the new characters we see in the pattern.

The pattern in made up of atoms.

Individual Characters

Atom	Description
a-z, A-Z, 0-9 and some punctution	These are ordinary characters that match themselves
"."	The dot, or period. This matches any single character execpt for the newline.

Character Classes

A group of characters to that are allowed to be matched one time. There are a few predefined classes, symbol that means a series of characters.

Atom	Description
`[ ]`	A bracketed list of characters, like `[GA]`. This indicates a single character can match any character in the bracketed list.
`\d`	Digits. Also can be written `[0-9]`
`\D`	Not digits. Also can be written`[^0-9]`
`\w`	Word character. Also can be written `[A-Za-z0-9_]` Note underscore is part of this class
`\W`	Not a word character, or `[^A-Za-z0-9_]`
`\s`	White space character. Also can be written `[ \r\t\n]`. Note the space character after the first `[`

Atom	Description
`\S`	Not whitespace. Also `[^ \r\\t\n]`

Anchors

A pattern can be anhcored to a region in the string

Atom	Description
`^`	Matches the beginning of the string
`$`	Matches the end of the string
`\b`	Matches a word boundary between `\w` and `\W`

Examples:

```
1 g..t
```

matches “gaat”, “goat”, and “gotta get a goat” (twice)

```
1 g[gatc][gatc]t
```

matches “gaat”, “gttt”, “gatt”, and “gotta get an agatt” (once)

```
1 \d\d\d-\d\d\d\d`
```

matches 376-8380, and 5128-8181 but not 055-98-2818.

```
1 ^\d\d\d-\d\d\d\d
```

matches 376-8380 and 376-83801 but not 5128-8181.

```
1 ^\d\d\d-\d\d\d\d$
```

only matches telephone numbers (without area code)

```
\bcat
```

matches “cat”, “catsup” and “more catsup please” but not “scat”.

```
\bcat\b
```

only text containing the word “cat”.

Quantifiers

Quantifiers quantify how many atoms are to be found. By default an atom matches only once. This behaviour can be modified following an atom with a quantifier.

Quantifier	Description
`?`	atom matches zero or exactly once
`*`	atom matches zero or more times
`+`	atom matches one or more times
`{3}`	atom matches exactly 3 times
`{2,4}`	atom matches between 2 and 4 times, inclusive
`{4,}`	atom matches at least 4 times

Examples:

```
goa?t
```

matches “goat” and “got”. Also any text that contains these words.

```
g.+t
```

matches “goat”, “goot”, and “grant”, among others.

```
g.*t
```

matches “gt”, “goat”, “goot”, and “grant”, among others.

```
^\d{3}-\d{4}$
```

matches US telephone numbers (no extra text allowed).

Something to think about.
1) What would be a pattern to recogize an email address? 2) What would be a pattern to recogize the ID portion of a sequence record in a FASTA file?

Variables and Patterns

Variables can be used to store patterns.

```
>>> pattern = r"[GA]C.?G"
>>> len (re.findall(pattern,dna))
7
```

In this example we stored our methylation pattern in the variable named 'pattern' and used it as the first argument to findall.

Either Or

A pipe '|' can be used to indicated that either the pattern before or after the '|' can match. Enclose the the two options in parenthesis.

```
big bad (wolf|sheep)
```

This pattern must match a string that contains "big" followed by a space followed by "bad" followed by a space followed by *either* "wolf" or "sheep" This would match, "big bad wolf" Or "big bad sheep"

Something to think about. 1) What would a pattern to match 'ATG' followed by a C or a T look like?

Subpatterns

Subpatterns, or parts of the pattern enclosed in parenthesis can be extracted and stored for later use.

```
Who's afraid of the big bad w(.+)f
```

This pattern has only one subpattern (.+)

You can combine parenthesis and quantifiers to quantify entire subpatterns

```
Who's afraid of the big (bad )?wolf\?
```

> This matches “Who’s afraid of the big bad wolf?”. As well as “Who’s afraid of the big wolf?”. The ‘bad’ is optional, it can be present 0 or 1 times in our string. This also shows how to literally match special characters. Use a ” in to escape them.

Something to think about: How would you create a pattern to capture the ID in a sequence record of a FASTA file in a subpattern.

Example FASTA sequence record.

```
>ID Optional Descrption
SEQUENCE
SEQUENCE
SEQUENCE
```

Using Subpatterns Inside the Regular Expression Match

This is helpful when you want to find a subpattern and then match the contents again

Once a subpattern matches, you can refer to it within the same regular expression. The first subpattern becomes \1, the second \2, the third \3, and so on.

```
Who's afraid of the big bad w(.)\1f
```

> This would match “Who’s afraid of the big bad woof” “Who’s afraid of the big bad weef” “Who’s afraid of the big bad waaf” But Not, “Who’s afraid of the big bad wolf” Or, “Who’s afraid of the big bad wife”

In a similar vein,

```
\b(\w+)s love \1 food\b
```

> This pattern will match “dogs love dog food” But not “dogs love monkey food”.

```
>>> str="dogs love dog food"
>>> found=re.search(r"\b(\w+)s love \1 food\b", str)
>>> if found:
...   print(found.group(1))
... else:
...   print("Not Found")
...
dog
>>> str="cats love cat food"
>>> found=re.search(r"\b(\w+)s love \1 food\b", str)
>>> if found:
...   print(found.group(1))
... else:
...   print("Not Found")
...
cat
>>> str="dogs love monkey food"
>>> found=re.search(r"\b(\w+)s love \1 food\b", str)
>>> if found:
...   print(found.group(1))
... else:
...   print("Not Found")
...
Not Found
```

This pattern matches:
- “dogs love dog food” - “cats love cat food” - but not “dogs love monkey food”

Subpatterns and Greediness

By default, regular expressions are “greedy”. They try to match as much as they can. Use the quantifier ‘?’ to make the match not greedy. The not greedy match is called ‘lazy’

```
>>> str = 'The fox ate my box of doughnuts'
>>> found = re.search(r"(f.+x)",str)
>>> print(found.group(1))
fox ate my box
```

The pattern f.+x does not match what you might expect, it matches past ‘fox’ all the way out to ‘fox ate my box’.
The ‘.+’ id greedy As many characters as possible are found that are between the ‘f’ and the ‘x’.

Let’s make this match lazy by using ‘?’

```
>>> found = re.search(r"(f.+?x)",str)
>>> print(found.group(1))
fox
```

The match is now lazy and will only match 'fox'

Using Subpatterns Outside the Regular Expression Match

Using the captured subpattern in code that follows the regular expression.

Outside the regular expression match statement, the matched subpatterns can be access with the `group()` method.

Example:

```
>>> dna =
    'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCC
>>> found=re.search( r"(.{50})TATTAT(.{25})"  , dna )
>>> upstream = found.group(1))
>>> print(upstream)
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
>>> downstream = found.group(2))
>>  print(downstream)
CCGGTTTCCAAAGACAGTCTTCTAA
```

1) This pattern will recognize a consensus transcription start site (TATTAT)
2) And store the 50 base pairs upstream of the site
3) And the 25 base pairs downstream of the site

If you want to find the upstream and downstream sequence of ALL 'TATTAT' sites, use the `findall()` function.

```
>>>
    dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTC
>>> found = re.findall( r"(.{50})TATTAT(.{25})"  , dna )
>>> print(found)
[('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA',
    'CCGGTTTCCAAAGACAGTCTTCTAA'),
    ('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA',
    'CCGGTTTCCAAAGACAGTCTTCTAA')]
```

The subpatterns are stored in tuples within a list. More about this type of data structure later.

Another option for retrieving the upstream and downstream subpatters is to put the `findall()` in a for loop

```
>>> dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTC
>>> for (upstream, downstream) in re.findall(r"(.{50})TATTAT(.{25})"  , dna ):
...     print("upstream:" , upstream)
...     print("downstream:" , downstream)
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1) This code executes the `findall()` function once
2) The subpatterns are returned
3) The subpatterns are stored in the variables upstream and downstream
4) The for block of code is executed
5) The `findall()` searches again
6) A match is found
7) New subpatterns are returned
8) The for block of code gets executed again
9) The `findall()` searches again, but no match is found
10) The for loop ends

One other way to get this done is with the `finditer()` function in a for loop

```
>>> dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTC
>>> for match in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
...     print("upstream:" , match.group(1))
...     print("downstream:" , match.group(2))
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1) This code executes `finditer()` function once.

2) The match object is returned. A match object will have all the information about the match

3) In the for block we call the `group()` method on the first match object returned

4) We print out the first and second subpattern using the `group()` method

5) The `finditer()` function is executed a second time and a match is found

6) The second match object is returned

7) The second subpatterns are retrieved from the match object using the `group()` method

8) The `finditer()` functin is executed again, but no matches found, so the loop ends

FYI: `match()` function is another regular expression function that looks for patterns. It is similar to search but it only looks at the begining of the string for the pattern while `search()` looks in the entire string. Usually `search()` and `findall()` will be more useful.

Practical Example: Codons

Extracting codons from a string of DNA can be accomplished by using a subpattern in a `findall()` function. Remember the `findall()` function will return a list of the matches.

```
>>> dna = 'GTTGCCTGAAATGGCGGAACCTTGAA'
>>> codons = re.findall(r"(.{3})",dna)
>>> print(codons)
['GTT', 'GCC', 'TGA', 'AAT', 'GGC', 'GGA', 'ACC', 'TTG']
```

Or you can use a for loop to do something to each match.

```
>>> for codon in re.findall(r"(.{3})",dna):
...   print(codon)
...
GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG
>>>
```

`finditer()` would also work in this for loop. Each codon can be accessed by using the `group()` method.

Truth and Regular Expression Matches

The `search()`, `match()`, `findall()`, and `finditer()` can be used in conditional tests. If a match is not found an empty list or 'None' is returned. These both are False.

```
>>> found=re.search( r"(.{50})TATTATZ(.{25})"  , dna )
>>> if found:
...     print("found it")
... else:
...     print("not found")
...
not found
>>> print(found)
None
```

None is False so the else block is executed and "not found" is printed

Using Regular expressions in substitutions

Earlier we went over how to find an exact pattern and replace it using the `replace()` method. To find a pattern and make a replacement the regular expression `sub()` function is used. This function takes the pattern, the replacement, the string to be searched, the number of times to do the replacement, and flags.

```
>>> str = "Who's afraid of the big bad wolf?"
>>> re.sub(r'w.+f' , 'goat', str)
"Who's afraid of the big bad goat?"
>>> print(str)
Who's afraid of the big bad wolf?
```

The `sub()` function returns "Who's afraid of the big bad goat?"
The value of variable str has not been altered
The new string can be stored in a new variable for later use.

Let's save the returned new string in a variable

```
>>> str = "He had a wife."
>>> new_str = re.sub(r'w.+f' , 'goat', str)
>>> print(new_str)
He had a goate.
>>> print(str)
He had a wife.
```

The characters between ‘w’ and ‘f’ have been replaced with ‘goat’.
The new string is saved in new_str

Using Regular expressions in substitutions

Sometimes you want to find a pattern and use it in the replacement.

```
>>> str = "Who's afraid of the big bad wolf?"
>>> new_str = re.sub(r"(\w+) (\w+) wolf" , r"\2 \1 wolf" , str)
>>> print(new_str)
Who's afraid of the bad big wolf?
```

We found two words before ‘wolf’ and swapped the order. \2 refers to the second subpattern
\1 refers to the first subpattern

Something to think about.
How would you use regular expressions to find all occurances of ‘ATG’ and replace with ‘-M-’ in this sequence ‘GCAGAGGTGATGGACTCCGTAATGGCCAAATGACACGT’? #### Using subpatterns in the replacement

Sometimes you want to find a pattern and use it in the replacement.

```
>>> str = "Who's afraid of the big bad wolf?"
>>> new_str = re.sub(r"(\w+) (\w+) wolf" , r"\2 \1 wolf" , str)
>>> print(new_str)
Who's afraid of the bad big wolf?
```

We found two words before ‘wolf’ and swapped the order. \2 refers to the second subpattern
\1 refers to the first subpattern

Something to think about.
How would you use regular expressions to find all occurances of ‘ATG’ and replace with ‘-M-’ in this sequence ‘GCAGAGGTGATGGACTCCGTAATGGCCAAATGACACGT’?

Link to Python 6 Problem Set

Link to Python 7 Problem Set

... 'gene2' : { ... 'seq' : "CAAATG", ... 'desc' : 'something', ... 'len' : 6, ... 'nt_comp' : { ... 'A' : 3, ... 'T' : 1, ... 'G' : 1, ... 'C' : 1, ... } ... } ... } >>> genes {'gene1': {'nt_comp': {'C': 2, 'G': 1, 'A': 1, 'T': 2}, 'desc': 'something', 'len': 6, 'seq': 'TATGCC'}, 'gene2': {'nt_comp': {'C': 1, 'G': 1, 'A': 3, 'T': 1}, 'desc': 'something', 'len': 6, 'seq': 'CAAATG'}} >>> genes['gene2']['nt_comp'] {'C': 1, 'G': 1, 'A': 3, 'T': 1} "' > Here we store a gene name as the outter most key, with a second level of keys for qualities of the gene, like sequence, length, nucleotide composition. We can retrieve a quality by using the gene name and quality in conjuction.

There are also specific data table and frame handling libraries like Pandas.

Link to Python 8 Problem Set

Python 9

BioPython

BioPython Overview

BioPython Subtopic 1

BioPython Subtopic 2

Bioinformatic Analysis and Tools

Bioinformatic Analysis and Tools Overview

- What you want to do:
  - tools to do it

Sequence Search and Alignments

Assembly

DNA

RNA

NGS

Ontology