

1

Unix Shell 1: Introduction

Lab Objective: *Unix is a popular operating system that is commonly used for servers and the basis for most open source software. Using Unix for writing and submitting labs will develop a foundation for future software development. In this lab we explore the basics of the Unix shell, including how to navigate and manipulate files, access remote machines with Secure Shell, and use Git for basic version control.*

Unix was first developed by AT&T Bell Labs in the 1970s. In the 1990s, Unix became the foundation of the Linux and MacOSX operating systems. Most servers are Linux-based, so knowing how to use Unix shells allows us to interact with servers and other Unix-based machines.

A *Unix shell* is a program that takes commands from a user and executes those commands on the operating system. We interact with the shell through a *terminal* (also called a command line), a program that lets you type in commands and gives those commands to the shell for execution.

Note

Windows is not built off of Unix, but it does come with a terminal called PowerShell. This terminal uses a different command syntax. We will not cover the equivalent commands in the Windows terminal, but you could download a Unix-based terminal such as Git Bash or Cygwin to complete this lab on a Windows machine (you will still lose out on certain commands). Alternatively, Windows 10 now offers a Windows Subsystem for Linux, WSL, which is a Linux operating system downloaded onto Windows.

Note

For this lab we will be working in the `UnixShell1` directory provided with the lab materials. If you have not yet downloaded the code repository, follow steps 1 through 6 in the **Getting Started** guide found at <https://foundations-of-applied-mathematics.github.io/> before proceeding with this lab. Make sure to run the `download_data.sh` script as described in step 5 of **Getting Started**; otherwise you will not have the necessary files to complete this lab.

Basic Unix Shell

Shell Scripting

The following sections of the lab will explore several shell commands. You can execute these commands by typing these commands directly into a terminal. Sometimes, though, you will want to execute a more complicated sequence of commands, or make it easy to execute the same set of commands over and over again. In those cases, it is useful to create a *script*, which is a sequence of shell commands saved in a file. Then, instead of typing the commands individually, you simply have to run the script, and it takes care of running all the commands.

In this lab we will be running and editing a bash script. Bash is the most commonly used Unix shell and is the default shell installed on most Unix-based systems.

The following is a very simple bash script. The command `echo <string>` prints `<string>` in the terminal.

```
#!/bin/bash
echo "Hello World!"
```

The first line, `#!/bin/bash`, tells the computer to use the bash interpreter to run the script, and where this interpreter is located. The `#!` is called the *shebang* or *hashbang* character sequence. It is followed by the absolute path to the bash interpreter.

To run a bash script, type `bash <script name>` into the terminal. Alternatively, you can execute any script by typing `./<script name>`, but note that the script must contain executable permissions for this to work. (We will learn more about permissions later in the lab.)

```
$ bash hello_world.sh
Hello World!
```

Navigation

Typically, people navigate computers by clicking on icons to open folders and programs. In the terminal, instead of point and click we use typed commands to move from folder to folder. In the Unix shell, we call folders *directories*. The file system is a set of nested directories containing files and other directories.

You can picture the file system as an tree, with directories as branches. Smaller branches stem from bigger branches, and all bigger branches eventually stem from the root of the tree. Similarly, in the Unix file system we have a "root directory", where all other directories are nested in. We denote it by using a single slash (/). All absolute paths originate at the root directory, which means all absolute path strings begin with the / character.

Begin by opening a terminal. The text you see in the upper left of the terminal is called the *prompt*. Before you start creating or deleting files, you'll want to know where you are. To see what directory you are currently working in, type `pwd` into the prompt. This command stands for **p**rint **w**orking **d**irectory, and it prints out a string telling you your current location.

```
~$ pwd
/home/username
```

To see the all the contents of your current directory, type the command `ls`, list segments.

```
~$ ls
Desktop    Downloads    Public    Videos
Documents  Pictures
```

The command `cd`, **change directory**, allows you to navigate directories. To change to a new directory, type the `cd` command followed by the name of the directory to which you want to move (if you `cd` into a file, you will get an error). You can move up one directory by typing `cd ..`.

Two important directories are the root directory and the home directory. You can navigate to the home directory by typing `cd ~` or just `cd`. You can navigate to root by typing `cd /`.

Problem 1. To begin, open a terminal and navigate to the `UnixShell1/` directory provided with this lab. Use `ls` to list the contents. There should be a file called `Shell1.zip` and a script called `unixshell1.sh`.^a

Run `unixshell1.sh`. This script will do the following:

1. Unzip `Shell1.zip`, creating a directory called `Shell1/`
2. Remove any previously unzipped copies of `Shell1/`
3. Execute various shell commands, to be added in the next few problems in this lab
4. Create a compressed version of `Shell1/` called `UnixShell1.tar.gz`.
5. Remove any old copies of `UnixShell1.tar.gz`

Now, open the `unixshell1.sh` script in a text editor. Add commands to the script, within the section for Problem 1, to do the following:

- Change into the `Shell1/` directory.
- Print a string (without using `echo`) telling you the directory you are currently working in.

Test your commands by running the script again and checking that it prints a string ending in the location `Shell1/`.

^aIf the necessary data files are not in your directory, `cd` one directory up by typing `cd ..` and type `bash download_data.sh` to download the data files for each lab.

Documentation and Help

When you encounter an unfamiliar command, the terminal has several tools that can help you understand what it does and how to use it. Most commands have manual pages, which give information about what the command does, the syntax required to use it, and different options to modify the command. To open the manual page for a command, type `man <command>`. Some commands also have an option called `--help`, which will print out information similar to what is contained in the manual page. To use this option, type `<command> --help`.

```
$ man ls
```

LS(1)	User Commands	LS(1)
NAME	ls - list directory contents	
SYNOPSIS	ls [OPTION]... [FILE]...	
DESCRIPTION	List information about the FILES (the current directory by default).	
	-a, -- all do not ignore entries starting with .	

The `apropos <keyword>` command will list all Unix commands that have `<keyword>` contained somewhere in their manual page names and descriptions. For example, if you forget how to copy files, you can type in `apropos copy` and you'll get a list of all commands that have `copy` in their description.

Flags

When you use `man`, you will see a list of options such as `-a`, `-A`, `--author`, etc. that modify how a command functions. These are called *flags*. You can use one flag on a command by typing `<command> -<flag>`, like `ls -a`, or combine multiple flags by typing `<command> -<flag1><flag2>`, etc. as in `ls -alt`.

For example, sometimes directories contain hidden files, which are files whose names begin with a dot character like `.bash`. The `ls` command, by default, does not list hidden files. Using the `-a` flag specifies that `ls` should not ignore hidden files. Find more common flags for `ls` in Table 1.1.

Flags	Description
-a	Do not ignore hidden files and folders
-l	List files and folders in long format
-r	Reverse order while sorting
-R	Print files and subdirectories recursively
-s	Print item name and size
-S	Sort by size
-t	Sort output by date modified

Table 1.1: Common flags of the `ls` command.

```
$ ls
file1.py file2.py

$ ls -a
. .. file1.py file2.py .hiddenfile.py

$ ls -alt    # Multiple flags can be combined into one flag
total 8
```

```
drwxr-xr-x  2 c c 4096 Aug 14 10:08 .
-rw-r--r--  1 c c    0 Aug 14 10:08 .hiddenfile.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file2.py
-rw-r--r--  1 c c    0 Aug 14 10:08 file1.py
drwxr-xr-x 38 c c 4096 Aug 14 10:08 ..
```

Problem 2. Within the script, add a command using `ls` to print one list of the contents of `Shell11/` with the following criteria:

- Include hidden files and folders
- List the files and folders in long format (include the permissions, date last modified, etc.)
- Sort the output by file size (largest files first)

Test your command by entering it into the terminal within `Shell11/` or by running the script and checking for the desired output.

Manipulating Files and Directories

In this section we will learn how to create, copy, move, and delete files and folders. To create a text file, use `touch <filename>`. To create a new directory, use `mkdir <dir_name>`.

```
~$ cd Test/                # navigate to test directory

~/Test$ ls                 # list contents of directory
file1.py

~/Test$ mkdir NewDirectory # create a new empty directory

~/Test$ touch newfile.py   # create a new empty file

~/Test$ ls
file1.py  NewDirectory  newfile.py
```

To copy a file into a directory, use `cp <filename> <dir_name>`. When making a copy of a directory, use the `-r` flag to recursively copy files contained in the directory. If you try to copy a directory without the `-r`, the command will return an error.

Moving files and directories follows a similar format, except no `-r` flag is used when moving one directory into another. The command `mv <filename> <dir_name>` will move a file to a folder and `mv <dir1> <dir2>` will move the first directory into the second.

If you want to rename a file, use `mv <file_old> <file_new>`; the same goes for directories.

```
~/Test$ ls
file1.py  NewDirectory  newfile.py

~/Test$ mv newfile.py NewDirectory/    # move file into directory
```

```
~/Test$ cp file1.py NewDirectory/           # make a copy of file1 in directory
~/Test$ cd NewDirectory/
~/Test/NewDirectory$ mv file1.py newname.py # rename file1.py
~/Test/NewDirectory$ ls
newfile.py  newname.py
```

When deleting files, use `rm <filename>`, and when deleting a directory, use `rm -r <dir_name>`. The `-r` flag tells the terminal to recursively remove all the files and subfolders within the targeted directory.

If you want to make sure your command is doing what you intend, the `-v` flag tells `rm`, `cp`, or `mkdir` to print strings in the terminal describing what it is doing.

When your terminal gets too cluttered, use `clear` to clean it up.

```
~/Test/NewDirectory$ cd ..           # move one directory up
~/Test$ rm -rv NewDirectory/         # remove a directory and its contents
removed 'NewDirectory/newname.py'
removed 'NewDirectory/newfile.py'
removed directory 'NewDirectory/'

~/Test$ rm file1.py                  # remove a file
~/Test$ ls                           # directory is now empty
~/Test$
```

Commands	Description
<code>clear</code>	Clear the terminal screen
<code>cp file1 dir1</code>	Create a copy of <code>file1</code> and move it to <code>dir1/</code>
<code>cp file1 file2</code>	Create a copy of <code>file1</code> and name it <code>file2</code>
<code>cp -r dir1 dir2</code>	Create a copy of <code>dir1/</code> and all its contents into <code>dir2/</code>
<code>mkdir dir1</code>	Create a new directory named <code>dir1/</code>
<code>mkdir -p path/to/new/dir1</code>	Create <code>dir1/</code> and all intermediate directories
<code>mv file1 dir1</code>	Move <code>file1</code> to <code>dir1/</code>
<code>mv file1 file2</code>	Rename <code>file1</code> as <code>file2</code>
<code>rm file1</code>	Delete <code>file1</code> [-i, -v]
<code>rm -r dir1</code>	Delete <code>dir1/</code> and all items within <code>dir1/</code> [-i, -v]
<code>touch file1</code>	Create an empty file named <code>file1</code>

Table 1.2: File Manipulation Commands

Table 1.2 contains all the commands we have discussed so far. Commonly used flags for some commands are contained in square brackets; use `man` or `--help` to see what these mean.

Problem 3. Add commands to the `unixshell1.sh` script to make the following changes in `Shell1/`:

- Delete the `Audio/` directory along with all its contents
- Create `Documents/`, `Photos/`, and `Python/` directories

- Change the name of the **Random/** directory to **Files/**

Test your commands by running the script and then using `ls` within **Shell11/** to check that each directory was deleted, created, or changed correctly.

Wildcards

As we are working in the file system, there will be times that we want to perform the same command to a group of similar files. For example, you may need to move all text files within a directory to a new directory. Rather than copy each file one at a time, we can apply one command to several files using *wildcards*. We will use the `*` and `?` wildcards. The `*` wildcard represents any string and the `?` wildcard represents any single character. Though these wildcards can be used in almost every Unix command, they are particularly useful when dealing with files.

```
$ ls
File1.txt  File2.txt  File3.jpg  text_files

$ mv -v *.txt text_files/
File1.txt -> text_files/File1.txt
File2.txt -> text_files/File2.txt

$ ls
File3.jpg  text_files
```

See Table 1.3 for examples of common wildcard usage.

Command	Description
<code>*.txt</code>	All files that end with <code>.txt</code> .
<code>image*</code>	All files that have <code>image</code> as the first 5 characters.
<code>*py*</code>	All files that contain <code>py</code> in the name.
<code>doc*.txt</code>	All files of the form <code>doc1.txt</code> , <code>doc2.txt</code> , <code>docA.txt</code> , etc.

Table 1.3: Common uses for wildcards.

Problem 4. Within the **Shell11/** directory, there are many files. Add commands to the script to organize these files into directories using wildcards. Organize by completing the following:

- Move all the `.jpg` files to the **Photos/** directory
- Move all the `.txt` files to the **Documents/** directory
- Move all the `.py` files to the **Python/** directory

Working With Files

Searching the File System

There are two commands we can use for searching through our directories. The `find` command is used to find files or directories with a certain name; the `grep` command is used to find lines within files matching a certain string. When searching for a specific string, both commands allow wildcards within the string. You can use wildcards so that your search string matches a broader set of strings.

```
# Find all files or directories in Shell1/ called "final"
# -type f,d specifies to look for files and directories
# . specifies to look in the current directory

$ find . -name "final" -type f,d
$          # There are no files with the exact name "final" in Shell1/

$ find . -name "*final*" -type f,d
./Files/May/finals
./Files/May/finals/finalproject.py
```

```
# Find all within files in Documents/ containing "Mary"
# -r tells grep to search all files with Documents/
# -n tells grep to print out the line number (2)

$ Shell1$ grep -nr "Mary" Documents/
Documents/people.txt:2:female,Mary,31
```

Command	Description
<code>find dir1 -type f -name "word"</code>	Find all files in <code>dir1/</code> (and its subdirectories) called <code>word</code> (<code>-type f</code> is for files; <code>-type d</code> is for directories)
<code>grep "word" filename</code>	Find all occurrences of <code>word</code> within <code>filename</code>
<code>grep -nr "word" dir1</code>	Find all occurrences of <code>word</code> within the files inside <code>dir1/</code> (<code>-n</code> lists the line number; <code>-r</code> performs a recursive search)

Table 1.4: Commands using `find` and `grep`.

Table 1.4 contains basic syntax for using these two commands. There are many more variations of syntax for `grep` and `find`, however. You can use `man grep` and `man find` to explore other options for using these commands.

File Security and Permissions

A file has three levels of permissions associated with it: the permission to read the file, to write (modify) the file, and to execute the file. There are also three categories of people who are assigned permissions: the user (the owner), the group, and others.

You can check the permissions for `file1` using the command `ls -l <file1>`. Note that your output will differ from that printed below; this is purely an example.


```
$ ls -l
-rw-rw-r-- 1 username groupname 194 Aug  5 20:20 calc.py
drw-rw-r-- 1 username groupname 373 Aug  5 21:16 Documents
-rwxr-x--x 1 username groupname  27 Aug  5 20:22 mult.py
-rw-rw-r-- 1 username groupname 721 Aug  5 20:23 project.py
```

The first character of each line denotes the type of the item whether it be a normal file, a directory, a symbolic link, etc. The next nine characters denote the permissions associated with that file.

For example, look at the output for `mult.py`. The first character `-` denotes that `mult.py` is a normal file. The next three characters, `rw`, tell us the owner can read, write, and execute the file. The next three characters, `r-x`, tell us members of the same group can read and execute the file, but not edit it. The final three characters, `--x`, tell us other users can execute the file and nothing more.

Permissions can be modified using the `chmod` command. There are multiple notations used to modify permissions, but the easiest to use when we want to make small modifications to a file's permissions is *symbolic permissions* notation. See Table 1.5 for more examples of using symbolic permissions notation, as well as other useful commands for working with permissions.

```
$ ls -l script1.sh
total 0
-rw-r--r-- 1 c c 0 Aug 21 13:06 script1.sh

$ chmod u+x script1.sh      # add permission for user to execute
$ chmod o-r script1.sh      # remove permission for others to read
$ ls -l script1.sh
total 0
-rwxr----- 1 c c 0 Aug 21 13:06 script1.sh
```

Command	Description
<code>chmod u+x file1</code>	Add executing (x) permissions to user (u)
<code>chmod g-w file1</code>	Remove writing (w) permissions from group (g)
<code>chmod o-r file1</code>	Remove reading (r) permissions from other other users (o)
<code>chmod a+w file1</code>	Add writing permissions to everyone (a)
<code>chown</code>	change owner
<code>chgrp</code>	change group
<code>getfacl</code>	view all permissions of a file in a readable format.

Table 1.5: Symbolic permissions notation and other useful commands

Running Files

To run a file for which you have execution permissions, type the file name preceded by `./`.

```
$ ./hello.sh
bash: ./hello.sh: Permission denied

$ ls -l hello.sh
```

```
-rw-r--r-- 1 username groupname 31 Jul 30 14:34 hello.sh

$ chmod u+x hello.sh          # You can now execute the file

$ ./hello.sh
Hello World!
```

Problem 5. Within `Shell11/`, there is a script called `organize_photos.sh`. First, use `find` to locate the script. Once you know the file location, add commands to your script so that it completes the following tasks:

- Moves `organize_photos.sh` to `Scripts/`
- Adds executable permissions to the script for the user
- Runs the script

Test that the script has been executed by checking that additional files have been moved into the `Photos/` directory. Check that permissions have been updated on the script by using `ls -l`.

Accessing Remote Machines

At times you will find it useful to perform tasks on a remote computer or server, such as running a script that requires a large amount of computing power on a supercomputer or accessing a data file stored on another machine.

Secure Shell

Secure Shell (SSH) allows you to remotely access other computers or servers securely. SSH is a network protocol encrypted using public-key cryptography. It ensures that all communication between your computer and the remote server is secure and encrypted.

The system you are connecting to is called the *host*, and the system you are connecting from is called the *client*. The first time you connect to a host, you will receive a warning saying the authenticity of the host can't be established. This warning is a default, and appears when you are connecting to a host you have not connected to before. When asked if you would like to continue connecting, select yes.

When prompted for your password, type your password as normal and press enter. No characters will appear on the screen, but they are still being logged. Once the connection is established, there is a secure tunnel through which commands and files can be exchanged between the client and host. To end a secure connection, type `exit`.

```
alice@mycomputer:~$ ssh alice27@acme01.byu.edu

alice27@acme01.byu.edu password:# Type password as normal
last login 7 Sept 11
```

```
[alice27@byu.local@acme01 ~]$ ls      # Commands are executed on the host
myacmeshare/

[alice27@byu.local@acme01 ~]$ exit  # End a secure connection
logout
Connection to acme01.byu.edu closed.

alice@mycomputer:~$                  # Commands are executed on the client
```

Secure Copy

To copy files from one computer to another, you can use the Unix command `scp`, which stands for secure copy protocol. The syntax for `scp` is essentially the same as the syntax for `cp`.

To copy a file from your computer to a specific location on a remote machine, use the syntax `scp <file1> <user@remote_host:file_path>`. As with `cp`, to copy a directory and all of its contents, use the `-r` flag.

```
# Make copies of file1 and dir2 in the home directory on acme01.byu.edu
alice@mycomputer:~$ scp file1 alice27@acme01.byu.edu:~/
alice@mycomputer:~$ scp -r dir1/dir2 alice27@acme01.byu.edu:~/
```

Use the syntax `scp -r <user@remote_host:file_path/dir1> <file_path>` to copy `dir1` from a remote machine to the location specified by `file_path` on your current machine.

```
# Make a local copy of dir1 (from acme01.byu.edu) in the home directory
alice@mycomputer:~$ scp -r alice27@acme01.byu.edu:~/dir1 ~
```

Commands	Description
<code>ssh username@remote_host</code>	Establish a secure connection with <code>remote_host</code>
<code>scp file1 user@remote_host:file_path/</code>	Create a copy of <code>file1</code> on host
<code>scp -r dir1 user@remote_host:file_path/</code>	Create a copy of <code>dir1</code> and its contents on host
<code>scp user@remote_host:file_path/file1 file_path2</code>	Create a local copy of file on client

Table 1.6: Basic syntax for `ssh` and `scp`.

Problem 6. The computer with host name `acme20.byu.edu` contains a file called `img_649.jpg`. Secure copy this file to your `UnixShell11/` directory. (Do not add the `scp` command to the script).

To `ssh` or `scp` to this computer, your username is your Net ID, and your password is your typical Net ID password. To use `scp` or `ssh` to access this computer, you will have to be on campus using BYU Wifi (or eduroam), or you can install a BYU specific VPN as explained at https://it.byu.edu/it?id=kb_article&sys_id=32da18c2877ad110df3c635d0ebb3551.

After secure copying, add a command to your script to copy the file from `UnixShell11/` into the directory `Shell11/Photos/`. (Make sure to leave a copy of the file in `UnixShell11/`,

otherwise the file will be deleted when you run the script again.)

Hint: To use `scp`, you will need to know the location of the file on the remote computer. Consider using `ssh` to access the machine and using `find`. The file is located somewhere in the directory `/sshlab`. Sometimes after logging onto the machine with `ssh`, there will appear to not be any directories you can access. Using the command `cd /` will fix this. When logging on initially, you also may get a message about not having a `myacmeshare`; this is not needed for this lab and the message may be ignored safely.

Git

Git is a version control system, meaning that it keeps a record of changes in a file. *Git* also facilitates collaboration between people working on the same code. It does both these things by managing updates between an online code repository and copies of the repository, called *clones*, stored locally on computers.

We will be using *git* to submit labs and return feedback on those labs. If *git* is not already installed on your computer, download it at <http://git-scm.com/downloads>.

Using Git

Git manages the history of a file system through *commits*, or checkpoints. Each time a new commit is added to the online repository, a checkpoint is created so that if need be, you can use or look back at an older version of the repository. You can use `git log` to see a list of previous commits. You can also use `git status` to see the files that have been changed in your local repository since the last commit.

Before making your own changes, you'll want to add any commits from other clones into your local repository. To do this, use the command `git pull origin master`.

Once you have made changes and want to make a new commit, there are normally three steps. To save these changes to the online repository, first add the changed files to the *staging area*, a list of files to save during the next commit, with `git add <filename(s)>`. If you want to add all changes that you have made to tracked files (files that are already included in the online repository), use `git add -u`.

Next, save the changes in the staging area with `git commit -m "<A brief message describing the changes>"`.

Finally, add the changes in this commit to the online repository with `git push origin master`.

```
$ cd MyDirectory/           # Navigate into a cloned repository
$ git pull origin master     # Pull new commits from online repository

### Make changes to file1.py ###

$ git add file1.py          # Add file to staging area
$ git commit -m "Made changes" # Commit changes in staging area
$ git push origin master     # Push changes to online repository
```

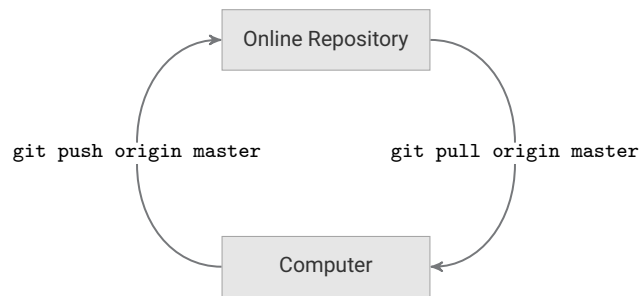


Figure 1.1: Exchanging git commits between the repository and a local clone.

Merge Conflicts

Git maintains order by raising an alert when changes are made to the same file in different clones and neither clone contains the changes made in the other. This is called a *merge conflict*, which happens when someone else has pushed a commit that you do not yet have, while you have also made one or more commits locally that they do not have.

Achtung!

When pulling updates with `git pull origin master`, your terminal may sometimes display the following merge conflict message.

```

Merge branch 'master' of https://bitbucket.org/<name>/<repo> into master
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
~
  
```

This screen, displayed in vim ([https://en.wikipedia.org/wiki/Vim_\(text_editor\)](https://en.wikipedia.org/wiki/Vim_(text_editor))), is asking you to enter a message to create a merge commit that will reconcile both changes. If you do not enter a message, a default message is used. To close this screen and create the merge commit with the default message, type `:wq` (the characters will appear in the bottom left corner of the terminal) and press `enter`.

Note

Vim is a terminal text editor available on essentially any computer you will use. When working with remote machines through `ssh`, vim is often the only text editor available to use. To exit vim, press `esc:wq`. To learn more about vim, visit the official documentation at <https://vimhelp.org>.

Command	Explanation
<code>git status</code>	Display the staging area and untracked changes.
<code>git pull origin master</code>	Pull changes from the online repository.
<code>git push origin master</code>	Push changes to the online repository.
<code>git add <filename(s)></code>	Add a file or files to the staging area.
<code>git add -u</code>	Add all modified, tracked files to the staging area.
<code>git commit -m "<message>"</code>	Save the changes in the staging area with a given message.
<code>git checkout <filename></code>	Revert changes to an unstaged file since the last commit.
<code>git reset HEAD <filename></code>	Remove a file from the staging area, but keep changes.
<code>git diff <filename></code>	See the changes to an unstaged file since the last commit.
<code>git diff --cached <filename></code>	See the changes to a staged file since the last commit.
<code>git config --local <option></code>	Record your credentials (<code>user.name</code> , <code>user.email</code> , etc.).

Table 1.7: Common git commands.

Problem 7. Using git commands, push `unixshell1.sh` and `UnixShell1.tar.gz` to your on-line git repository. Do not add anything else in the `UnixShell1/` directory to the online repository.

2 Introduction to Python

Lab Objective: *Python is a powerful general-purpose programming language. It can be used interactively, allowing for very rapid development. Python has many powerful scientific computing tools, making it an ideal language for applied and computational mathematics. In this introductory lab we introduce Python syntax, data types, functions, and control flow tools. These Python basics are an essential part of almost every problem you will solve and almost every program you will write.*

Getting Started

Python is quickly gaining momentum as a fundamental tool in scientific computing. *Anaconda* is a free distribution service by Continuum Analytics, Inc., that includes the cross-platform Python *interpreter* (the software that actually executes Python code) and many Python libraries that are commonly used for applied and computational mathematics. To install Python via Anaconda, go to <https://www.anaconda.com/download/>, download the installer for **Python 3.10** corresponding to your operating system, and follow the on-screen instructions.

Running Python

Python files are saved with a `.py` extension. For beginners, we strongly recommend using a simple text editor for writing Python files, though many free IDEs (Integrated Development Environments—large applications that facilitate code development with some sophisticated tools) are also compatible with Python. For now, the simpler the coding environment, the better.

A plain Python file looks similar to the following code.

```
# filename.py
"""This is the file header.
The header contains basic information about the file.
"""

if __name__ == "__main__":
    pass                                # 'pass' is a temporary placeholder.
```

The `#` character creates a single-line *comment*. Comments are ignored by the interpreter and serve as annotations for the accompanying source code. A pair of three quotes, `""" """` or `''' '''`, creates a multi-line string literal, which may also be used as a multi-line comment. A triple-quoted string literal at the top of the file serves as the *header* for the file. The header typically identifies the author and includes instructions on using the file. Executable Python code comes after the header.

Problem 1. Open the file named `python_intro.py` (or create the file in a text editor if you don't have it). Add your information to the header at the top, then add the following code.

```
if __name__ == "__main__":
    print("Hello, world!")           # Indent with four spaces (NOT a tab).
```

Open a command prompt (*Terminal* on Linux or Mac and *Command Prompt* or *GitBash* on Windows) and navigate to the directory where the new file is saved. Use the command `ls` (or `DIR` on Windows) to list the files and folders in the current directory, `pwd` (`CD` , on Windows) to print the working directory, and `cd` to change directories.

```
$ pwd                               # Print the working directory.
/Users/Guest
$ ls                                # List the files and folders here.
Desktop    Documents    Downloads    Pictures    Music
$ cd Documents                       # Navigate to a different folder.
$ pwd
/Users/Guest/Documents
$ ls                                # Check to see that the file is here.
python_intro.py
```

Now the Python file can be executed with the following command:

```
$ python python_intro.py
```

If `Hello, world!` is displayed on the screen, you have just successfully executed your first Python program!

Achtung!

The `if __name__ == "__main__"` clause is incredibly helpful to create test functions and debug your code. In order to use `if __name__ == "__main__"` to test declared functions, **it must be placed at the end of your file**. If you attempt to run that block of code at the beginning of the file with functions that are declared afterwards, you'll get an error of having undefined functions.

IPython

Python can be run interactively using several interfaces. The most basic of these is the Python interpreter. In this and subsequent labs, the triple brackets `>>>` indicate that the given code is being executed one line at a time via the Python interpreter.

```
$ python                                # Start the Python interpreter.
>>> print("This is plain Python.")      # Execute some code.
This is plain Python.
```

There are, however, more useful interfaces. Chief among these is *IPython*,¹ `[?, ?]` which is included with the Anaconda distribution. To execute a script in IPython, use the `%run` command.

```
>>> exit()                              # Exit the Python interpreter.
$ ipython                                # Start IPython.

In [1]: print("This is IPython!")        # Execute some code.
This is IPython!

In [2]: %run python_intro.py             # Run a particular Python script.
Hello, world!
```

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection quickly reveals all methods and attributes associated with an object. IPython also has a built-in `help()` function that provides interactive help.

```
# A list is a basic Python data structure. To see the methods associated with
# a list, type the object name (list), followed by a period, and press tab.
In [1]: list.    # Press 'tab'.
        append() count()  insert()  remove()
        clear()  extend() mro()     reverse()
        copy()   index()  pop()     sort()

# To learn more about a specific method, use a '?' and hit 'Enter'.
In [1]: list.append?
Signature: list.append(self, object, /)
Docstring: Append object to the end of the list.
Type:      method_descriptor

In [2]: help()                                # Start IPython's interactive help utility.

help> list                                # Get documentation on the list class.
Help on class list in module builtins:

class list(object)
| list(iterable=(),/)
| # ...                                     # Press 'q' to exit the info screen.
```

¹See <https://ipython.org/> and <https://jupyter.org/>.

```
help> quit                                # End the interactive help session.
```

Note

Use IPython side-by-side with a text editor to test syntax and small code snippets quickly. Testing small pieces of code in IPython **before** putting it into a program reveals errors and greatly speeds up the coding process. Consult the internet with questions; stackoverflow.com is a particularly valuable resource for answering common programming questions.

The best way to learn a new coding language is by actually writing code. Follow along with the examples in the yellow code boxes in this lab by executing them in an IPython console. Avoid copy and paste for now; your fingers need to learn the language as well.

Python Basics

Arithmetic

Python can be used as a calculator with the regular +, -, *, and / operators. Use ** for exponentiation and % for modular division.

```
>>> 3**2 + 2*5                                # Python obeys the order of operations.
19

>>> 13 % 3                                    # The modulo operator % calculates the
1                                             # remainder: 13 = (3*4) + 1.
```

In most Python interpreters, the underscore character _ is a variable with the value of the previous command's output, like the ANS button on many calculators.

```
>>> 12 * 3
36
>>> _ / 4
9.0
```

Data comparisons like < and > act as expected. The == operator checks for numerical equality and the <= and >= operators correspond to \leq and \geq , respectively. To connect multiple boolean expressions, use the operators [and](#), [or](#), and [not](#).²

```
>>> 3 > 2.99
True
>>> 1.0 <= 1 or 2 > 3
True
>>> 7 == 7 and not 4 < 4
True
```

²In many other programming languages, the [and](#), [or](#), and [not](#) operators are written as &&, ||, and !, respectively. Python's convention is much more readable and does not require parentheses.

```
>>> True and True and True and True and True and False
False
>>> False or False or False or False or False or True
True
>>> True or not True
True
```

Variables

Variables are used to temporarily store data. A **single** equals sign = assigns one or more values (on the right) to one or more variable names (on the left). A **double** equals sign == is a comparison operator that returns **True** or **False**, as in the previous code block.

Unlike many programming languages, Python does not require a variable's data type to be specified upon initialization. Because of this, Python is called a *dynamically typed* language.

```
>>> x = 12                                # Initialize x with the integer 12.
>>> y = 2 * 6                             # Initialize y with the integer 2*6 = 12.
>>> x == y                               # Compare the two variable values.
True

>>> x, y = 2, 4                           # Give both x and y new values in one line.
>>> x == y
False
```

Functions

To define a function, use the **def** keyword followed by the function name, a parenthesized list of parameters, and a colon. Then indent the function body using exactly **four** spaces.

```
>>> def add(x, y):
...     return x + y                     # Indent with four spaces.
```

Achtung!

Many other languages use the curly braces {} to delimit blocks, but Python uses whitespace indentation. In fact, whitespace is essentially the only thing that Python is particularly picky about compared to other languages: **mixing tabs and spaces confuses the interpreter and causes problems**. Most text editors have a setting to set the indentation type to spaces so you can use the tab key on your keyboard to insert four spaces (sometimes called soft tabs). For consistency, **never** use tabs; **always** use spaces.

Functions are defined with *parameters* and called with *arguments*, though the terms are often used interchangeably. Below, **width** and **height** are parameters for the function **area()**. The values 2 and 5 are the arguments that are passed when calling the function.

```
>>> def area(width, height):          # Define the function.
...     return width * height
...
>>> area(2, 5)                       # Call the function.
10
```

Python functions can also return multiple values.

```
>>> def arithmetic(a, b):
...     return a - b, a * b          # Separate return values with commas.
...
>>> x, y = arithmetic(5, 2)          # Unpack the returns into two variables.
>>> print(x, y)
3 10
```

The keyword `lambda` is a shortcut for creating one-line functions. For example, the polynomials $f(x) = 6x^3 + 4x^2 - x + 3$ and $g(x, y, z) = x + y^2 - z^3$ can be defined as functions in one line each.

```
# Define the polynomials the usual way using 'def'.
>>> def f(x):
...     return 6*x**3 + 4*x**2 - x + 3
>>> def g(x, y, z):
...     return x + y**2 - z**3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> f = lambda x: 6*x**3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z**3
```

Note

Documentation is important in every programming language. Every function should have a docstring—a string literal in triple quotes just under the function declaration—that describes the purpose of the function, the expected inputs and return values, and any other notes that are important to the user. Short docstrings are acceptable for very simple functions, but more complicated functions require careful and detailed explanations.

```
>>> def add(x, y):
...     """Return the sum of the two inputs."""
...     return x + y

>>> def area(width, height):
...     """Return the area of the rectangle with the specified width
...     and height.
...     """
...     return width * height
... 
```

```
>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Lambda functions cannot have custom docstrings, so the `lambda` keyword should be only be used as a shortcut for very simple or intuitive functions that need no additional labeling.

Problem 2. The volume of a sphere with radius r is $V = \frac{4}{3}\pi r^3$. In your Python file from Problem 1, define a function called `sphere_volume()` that accepts a single parameter r . Return the volume of the sphere of radius r , using 3.14159 as an approximation for π (for now). Also write an appropriate docstring for your function.

To test your function, call it under the `if __name__ == "__main__"` clause and print the returned value. Run your file to see if your answer is what you expect it to be.

Achtung!

The `return` statement instantly ends the function call and passes the return value to the function caller. However, functions are not required to have a return statement. A function without a return statement implicitly returns the Python constant `None`, which is similar to the special value `null` of many other languages. Calling `print()` at the end of a function does **not** cause a function to return any values.

```
>>> def oops(i):
...     """Increment i (but forget to return anything)."""
...     print(i + 1)
...
>>> def increment(i):
...     """Increment i."""
...     return i + 1
...
>>> x = oops(1999)           # x contains 'None' since oops()
2000                        # doesn't have a return statement.
>>> y = increment(1999)      # However, y contains a value.
>>> print(x, y)
None 2000
```

If you have any intention of using the results of a function, use a `return` statement.

It is also possible to specify *default values* for a function's parameters. In the following example, the function `pad()` has three parameters, and the value of `c` defaults to 0. If it is not specified in the function call, the variable `c` will contain the value 0 when the function is executed.

```
>>> def pad(a, b, c=0):
```

```

...     """Print the arguments, plus an zero if c is not specified."""
...     print(a, b, c)
...
>>> pad(1, 2, 3)                # Specify each parameter.
1 2 3
>>> pad(1, 2)                   # Specify only non-default parameters.
1 2 0

```

Arguments are passed to functions based on position or name, and positional arguments must be defined before named arguments. For example, `a` and `b` must come before `c` in the function definition of `pad()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

```

# Try defining printer with a named argument before a positional argument.
>>> def pad(c=0, a, b):
...     print(a, b, c)
...
SyntaxError: non-default argument follows default argument

```

```

# Correctly define pad() with the named argument after positional arguments.
>>> def pad(a, b, c=0):
...     """Print the arguments, plus an zero if c is not specified."""
...     print(a, b, c)
...

# Call pad() with 3 positional arguments.
>>> pad(2, 4, 6)
2 4 6

# Call pad() with 3 named arguments. Note the change in order.
>>> pad(b=3, c=5, a=7)
7 3 5

# Call pad() with 2 named arguments, excluding c.
>>> pad(b=1, a=2)
2 1 0

# Call pad() with 1 positional argument and 2 named arguments.
>>> pad(1, c=2, b=3)
1 3 2

```

Problem 3. The built-in `print()` function has the useful keyword arguments `sep` and `end`. It accepts any number of positional arguments and prints them out with `sep` inserted between values (defaulting to a space), then prints `end` (defaulting to the *newline character* `'\n'`).

Write a function called `isolate()` that accepts five arguments. The function should print

the first three arguments separated by 5 spaces and then print the last two arguments with a single space separating the last three arguments. For example,

```
>>> isolate(1, 2, 3, 4, 5)
1      2      3 4 5
```

Achtung!

In previous versions of Python, `print()` was a statement (like `return`), not a function, and could therefore be executed without parentheses. However, it lacked keyword arguments like `sep` and `end`. If you are using Python 2.7, include the following line at the top of the file to turn the `print` statement into the new `print()` function.

```
>>> from __future__ import print_function
```

Data Types and Structures

Numerical Types

Python has four numerical data types: `int`, `long`, `float`, and `complex`. Each stores a different kind of number. The built-in function `type()` identifies an object's data type.

```
>>> type(3)                                # Numbers without periods are integers.
int

>>> type(3.0)                             # Floats have periods (3. is also a float).
float
```

Python has two types of division: integer and float. The `/` operator performs float division (true fractional division), and the `//` operator performs integer division, which rounds the result down to the next integer. If both operands for `//` are integers, the result will be an `int`. If one or both operands are floats, the result will be a `float`. Regular division with `/` always returns a `float`.

```
>>> 15 / 4                                # Float division performs as expected.
3.75

>>> 15 // 4                              # Integer division rounds the result down.
3

>>> 15. // 4
3.0
```

Achtung!

In previous versions of Python, using `/` with two integers performed integer division, even in cases where the division was not even. This can result in some incredibly subtle and frustrating errors. If you are using Python 2.7, always include a `.` on the operands or cast at least one as a float when you want float division.

```
# PYTHON 2.7
>>> 15 / 4
3
# The answer should be 3.75, but the
# interpreter does integer division!

>>> 15. / float(4)
3.75
# 15. and float(4) are both floats, so
# the interpreter does float division.
```

Alternatively, including the following line at the top of the file redefines the `/` and `//` operators so they are handled the same way as in Python 3.

```
>>> from __future__ import division
```

Python also supports complex numbers computations by pairing two numbers as the real and imaginary parts. Use the letter *j*, not *i*, for the imaginary part.

```
>>> x = complex(2,3)
# Create a complex number this way...
>>> y = 4 + 5j
# ...or this way, using j (not i).
>>> x.real
2.0
# Access the real part of x.
>>> y.imag
5.0
# Access the imaginary part of y.
```

Strings

In Python, strings are created with either single or double quotes. To concatenate two or more strings, use the `+` operator between string variables or literals.

```
>>> str1 = "Hello"
>>> str2 = 'world'
>>> my_string = str1 + " " + str2 + '!'
>>> my_string
'Hello world!'
```

Parts of a string can be accessed using *slicing*, indicated by square brackets `[]`. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 1.

```
>>> my_string = "Hello world!"
```



```

>>> my_string[4]                # Indexing begins at 0.
'o'
>>> my_string[-1]               # Negative indices count backward from the end.
'!'

# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
'Hello'

# Slice from the 6th character to the end.
>>> my_string[6:]
'world!'

# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'

# Get every other character in the string.
>>> my_string[::2]
'Hlowrd'

```

Problem 4. Write two new functions, called `first_half()` and `backward()`.

1. `first_half()` should accept a parameter and return the first half of it, excluding the middle character if there is an odd number of characters.
(Hint: the built-in function `len()` returns the length of the input.)
2. The `backward()` function should accept a parameter and reverse the order of its characters using slicing, then return the reversed parameter.
(Hint: The `step` parameter used in slicing can be negative.)

Use IPython to quickly test your syntax for each function.

Lists

A Python `list` is created by enclosing comma-separated values with square brackets `[]`. Entries of a list do **not** have to be of the same type. Access entries in a list with the same indexing or slicing operations used with strings.

```

>>> my_list = ["Hello", 93.8, "world", 10]
>>> my_list[0]
'Hello'
>>> my_list[-2]
'world'
>>> my_list[:2]
['Hello', 93.8]

```

Common list methods (functions) include `append()`, `insert()`, `remove()`, and `pop()`. Consult IPython for details on each of these methods using object introspection.

```
>>> my_list = [1, 2]                # Create a simple list of two integers.
>>> my_list.append(4)               # Append the integer 4 to the end.
>>> my_list.insert(2, 3)            # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)               # Remove 3 from the list.
>>> my_list.pop()                  # Remove (and return) the last entry.
4
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"             # 'in' also works on strings.
False
```

Tuples

A Python `tuple` is an ordered collection of elements, created by enclosing comma-separated values with parentheses (and). Tuples are similar to lists, but they are much more rigid, have less built-in operations, and cannot be altered after creation. Lists are therefore preferable for managing dynamic ordered collections of objects.

When multiple objects are returned by a function, they are returned as a tuple. For example, recall that the `arithmetic()` function returns two values.

```
>>> x, y = arithmetic(5,2)          # Get each value individually,
>>> print(x, y)
3 10
>>> both = arithmetic(5,2)          # or get them both as a tuple.
>>> print(both)
(3, 10)
```

Problem 5. Write a function called `list_ops()`. Define a list with the entries `"bear"`, `"ant"`, `"cat"`, and `"dog"`, in that order. Then perform the following operations on the list:

1. Append `"eagle"`.
2. Replace the entry at index 2 with `"fox"`.
3. Remove (or pop) the entry at index 1.
4. Sort the list in reverse alphabetical order.
5. Replace `"eagle"` with `"hawk"`.
(Hint: the list's `index()` method may be helpful.)
6. Add the string `"hunter"` to the last entry in the list.

Return the resulting list of strings.

Work out (on paper) what the result should be, then check that your function returns the correct list. Consider printing the list at each step to see the intermediate results.

Sets

A Python `set` is an unordered collection of distinct objects. Objects can be added to or removed from a set after its creation. Initialize a set with curly braces `{ }`, separating the values by commas, or use `set()` to create an empty set. Like mathematical sets, Python sets have operations like union, intersection, difference, and symmetric difference.

```
# Initialize some sets. Note that repeats are not added.
>>> gym_members = {"Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"}
>>> print(gym_members)
{'Doe, John', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.add("Lytle, Josh")      # Add an object to the set.
>>> gym_members.discard("Doe, John")    # Delete an object from the set.
>>> print(gym_members)
{'Lytle, Josh', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Ian", "Webb, Jared"})
{'Lytle, Josh'}
>>> gym_members.difference({"Brown, Bob", "Sharp, Sarah"})
{'Lytle, Josh', 'Smith, Jane'}
```

Dictionaries

Like a set, a Python `dict` (dictionary) is an unordered data type. A dictionary stores key-value pairs, called *items*. The values of a dictionary are indexed by its keys. Dictionaries are initialized with curly braces, colons, and commas. Use `dict()` or `{}` to create an empty dictionary.

```

>>> my_dictionary = {"business": 4121, "math": 2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
2061

# Add a value indexed by 'science' and delete the 'business' keypair.
>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")      # Use 'pop' or 'popitem' to remove.
4121
>>> print(my_dictionary)
{'math': 2061, 'visual arts': 7321, 'science': 6284}

# Display the keys and values.
>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])

```

As far as data access goes, lists are like dictionaries whose keys are the integers $0, 1, \dots, n-1$, where n is the number of items in the list. The keys of a dictionary need not be integers, but they must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the Standard Library lab.

Type Casting

The names of each of Python's data types can be used as functions to cast a value as that type. This is particularly useful for converting between integers and floats.

```

# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)
>>> z = complex(3)
>>> print(x, y, z)
3 3.0 (3+0j)

# Cast a list as a set and vice versa.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
>>> str(list(set([complex(float(3))])))
'[(3+0j)]'

```

Control Flow Tools

Control flow blocks dictate the order in which code is executed. Python supports the usual control flow statements including `if` statements, `while` loops and `for` loops.

The If Statement

An `if` statement executes the indented code **if** (and only if) the given condition holds. The `elif` statement is short for “else if” and can be used multiple times following an if statement, or not at all. The `else` keyword may be used at most once at the end of a series of `if/elif` statements.

```
>>> food = "bagel"
>>> if food == "apple":           # As with functions, the colon denotes
...     print("72 calories")      # the start of each code block.
... elif food == "banana" or food == "carrot":
...     print("105 calories")
... else:
...     print("calorie count unavailable")
...
calorie count unavailable
```

Problem 6. Write a function called `pig_latin()`. Accept a string parameter `word`, translate it into Pig Latin, then return the translation. Specifically, if `word` starts with a vowel, add “hay” to the end; if `word` starts with a consonant, take the first character of `word`, move it to the end, and add “ay”. (Hint: use the `in` operator to check if the first letter is a vowel.)

The While Loop

A `while` loop executes an indented block of code **while** the given condition holds.

```
>>> i = 0
>>> while i < 10:
...     print(i, end=' ')          # Print a space instead of a newline.
...     i += 1                    # Shortcut syntax for i = i+1.
...
0 1 2 3 4 5 6 7 8 9
```

There are two additional useful statements to use inside of loops:

1. `break` manually exits the loop, regardless of which iteration the loop is on or if the termination condition is met.
2. `continue` skips the current iteration and returns to the top of the loop block if the termination condition is still not met.

```

>>> i = 0
>>> while True:
...     print(i, end=' ')
...     i += 1
...     if i >= 10:
...         break                # Exit the loop.
...
0 1 2 3 4 5 6 7 8 9

>>> i = 0
>>> while i < 10:
...     i += 1
...     if i % 3 == 0:
...         continue            # Skip multiples of 3.
...     print(i, end=' ')
1 2 4 5 7 8 10

```

The For Loop

A **for** loop iterates over the items in any *iterable*. Iterables include (but are not limited to) strings, lists, sets, and dictionaries.

```

>>> colors = ["red", "green", "blue", "yellow"]
>>> for entry in colors:
...     print(entry + "!")
...
red!
green!
blue!
yellow!

```

The **break** and **continue** statements also work in for loops, but a **continue** in a for loop will automatically increment the index or item, whereas a **continue** in a while loop makes no automatic changes to any variable.

```

>>> for word in ["It", "definitely", "looks", "pretty", "bad", "today"]:
...     if word == "definitely":
...         continue
...     elif word == "bad":
...         break
...     print(word, end=' ')
...
It looks pretty

```

In addition, Python has some very useful built-in functions that can be used in conjunction with the **for** statement:

1. `range(start, stop, step)`: Produces a sequence of integers, following slicing syntax. If only one argument is specified, it produces a sequence of integers from 0 to the argument, incrementing by one. This function is used **very** often.
2. `zip()`: Joins multiple sequences so they can be iterated over simultaneously.
3. `enumerate()`: Yields both a count and a value from the sequence. Typically used to get both the index of an item and the actual item simultaneously.
4. `reversed()`: Reverses the order of the iteration.
5. `sorted()`: Returns a new list of sorted items that can then be used for iteration.

Each of these functions except for `sorted()` returns an *iterator*, an object that is built specifically for looping but not for creating actual lists. To put the items of the sequence in a collection, use `list()`, `set()`, or `tuple()`.

```
# Strings and lists are both iterables.
>>> vowels = "aeiou"
>>> colors = ["red", "yellow", "white", "blue", "purple"]

# Iterate by index.
>>> for i in range(5):
...     print(i, vowels[i], colors[i])
...
0 a red
1 e yellow
2 i white
3 o blue
4 u purple

# Iterate through both sequences at once.
>>> for letter, word in zip(vowels, colors):
...     print(letter, word)
...
a red
e yellow
i white
o blue
u purple

# Get the index and the item simultaneously.
>>> for i, color in enumerate(colors): #
...     print(i, color)
...
0 red
1 yellow
2 white
3 blue
4 purple
```

```

# Iterate through the list in sorted (alphabetical) order.
>>> for item in sorted(colors):
...     print(item, end=' ')
...
blue purple red white yellow

# Iterate through the list backward.
>>> for item in reversed(colors):
...     print(item, end=' ')
...
purple blue white yellow red

# range() arguments follow slicing syntax.
>>> list(range(10))           # Integers from 0 to 10, exclusive.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(4, 8))         # Integers from 4 to 8, exclusive.
[4, 5, 6, 7]

>>> set(range(2, 20, 3))      # Every third integer from 2 to 20.
{2, 5, 8, 11, 14, 17}

```

Problem 7. This problem originates from <https://projecteuler.net>, an excellent resource for math-related coding problems.

A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$. Write a function called `palindrome()` that finds and returns the largest palindromic number made from the product of two 3-digit numbers.

List Comprehension

A *list comprehension* uses for loop syntax between square brackets to create a list. This is a powerful, efficient way to build lists. The code is concise and runs quickly.

```

>>> [float(n) for n in range(5)]
[0.0, 1.0, 2.0, 3.0, 4.0]

```

List comprehensions can be thought of as “inverted loops”, meaning that the body of the loop comes before the looping condition. The following loop and list comprehension produce the same list, but the list comprehension takes only about two-thirds the time to execute.

```

>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]

```


Tuple, set, and dictionary comprehensions can be done in the same way as list comprehensions by using the appropriate style of brackets on the end.

```
>>> colors = ["red", "blue", "yellow"]
>>> {c[0]:c for c in colors}
{'y': 'yellow', 'r': 'red', 'b': 'blue'}

>>> {"bright " + c for c in colors}
{'bright blue', 'bright red', 'bright yellow'}
```

Problem 8. The alternating harmonic series is defined as follows.

$$\sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots = \ln(2)$$

Write a function called `alt_harmonic()` that accepts an integer n . Use a list comprehension to quickly compute and sum the first n terms of this series (be careful not to sum only $n - 1$ terms). The sum of the first 500,000 terms of this series approximates $\ln(2)$ to five decimal places.

(Hint: consider using Python's built-in `sum()` function.)

Additional Material

Further Reading

Refer back to this and other introductory labs often as you continue getting used to Python syntax and data types. As you continue your study of Python, we strongly recommend the following readings.

- The official Python tutorial: <http://docs.python.org/3.6/tutorial/introduction.html> (especially chapters 3, 4, and 5).
- Section 1.2 of the SciPy lecture notes: <http://scipy-lectures.github.io/>.
- PEP8 - Python style guide: <http://www.python.org/dev/peps/pep-0008/>.

Generalized Function Input

On rare occasion, it is necessary to define a function without knowing exactly what the parameters will be like or how many there will be. This is usually done by defining the function with the parameters `*args` and `**kwargs`. Here `*args` is a list of the positional arguments and `**kwargs` is a dictionary mapping the keywords to their argument. This is the most general form of a function definition.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print("Argument " + str(i) + ":", arg)
...     for key in kwargs:
...         print("Keyword", key, "-->", kwargs[key])
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword missing --> True
Keyword exceptional --> False
```

See <https://docs.python.org/3.6/tutorial/controlflow.html> for more on this topic.

Function Decorators

A *function decorator* is a special function that “wraps” other functions. It takes in a function as input and returns a new function that pre-processes the inputs or post-processes the outputs of the original function.

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs) # Call the decorated function.
...         print("output type:", type(output)) # Process before finishing.
...         return output # Return the function output.
...     return wrapper
```

The outer function, `typewriter()`, returns the new function `wrapper()`. Since `wrapper()` accepts `*args` and `**kwargs` as arguments, the input function `func()` could accept any number of positional or keyword arguments.

Apply a decorator to a function by tagging the function's definition with an `@` symbol and the decorator name.

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b // c
```

Placing the tag above the definition is equivalent to adding the following line of code after the function definition:

```
>>> combine = typewriter(combine)
```

Now calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Function decorators can also be customized with arguments. This requires another level of nesting: the outermost function must define and return a decorator that defines and returns a wrapper.

```
>>> def repeat(times):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(3)
... def hello_world():
...     print("Hello, world!")
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

See <https://www.python.org/dev/peps/pep-0318/> for more details.

3

The Standard Library

Lab Objective: *Python is designed to make it easy to implement complex tasks with little code. To that end, every Python distribution includes several built-in functions for accomplishing common tasks. In addition, Python is designed to import and reuse code written by others. A Python file with code that can be imported is called a module. All Python distributions include a collection of modules for accomplishing a variety of tasks, collectively called the Python Standard Library. In this lab we explore some built-in functions, learn how to create, import, and use modules, and become familiar with the standard library.*

Built-in Functions

Python has several built-in functions that may be used at any time. IPython's object introspection feature makes it easy to learn about these functions: start IPython from the command line and use `?` to bring up technical details on each function.

```
In [1]: min?
```

```
Docstring:
```

```
min(iterable, *[, default=obj, key=func]) -> value
```

```
min(arg1, arg2, *args, *[, key=func]) -> value
```

```
With a single iterable argument, return its smallest item. The
default keyword-only argument specifies an object to return if
the provided iterable is empty.
```

```
With two or more arguments, return the smallest argument.
```

```
Type:          builtin_function_or_method
```

```
In [2]: len?
```

```
Signature: len(obj, /)
```

```
Docstring: Return the number of items in a container.
```

```
Type:          builtin_function_or_method
```

Function	Returns
<code>abs()</code>	The absolute value of a real number, or the magnitude of a complex number.
<code>min()</code>	The smallest element of a single iterable, or the smallest of several arguments. Strings are compared based on lexicographical order: numerical characters first, then upper-case letters, then lower-case letters.
<code>max()</code>	The largest element of a single iterable, or the largest of several arguments.
<code>len()</code>	The number of items of a sequence or collection.
<code>round()</code>	A float rounded to a given precision in decimal digits.
<code>sum()</code>	The sum of a sequence of numbers.

Table 3.1: Common built-in functions for numerical calculations.

```
# abs() can be used with real or complex numbers.
>>> print(abs(-7), abs(3 + 4j))
7 5.0

# min() and max() can be used on a list, string, or several arguments.
# String characters are ordered lexicographically.
>>> print(min([4, 2, 6]), min("aXbYcZ"), min('1', 'a', 'A'))
2 X 1
>>> print(max([4, 2, 6]), max("aXbYcZ"), max('1', 'a', 'A'))
6 c a

# len() can be used on a string, list, set, dict, tuple, or other iterable.
>>> print(len([2, 7, 1]), len("abcdef"), len({1, 'a', 'a'}))
3 6 2

# sum() can be used on iterables containing numbers, but not strings.
>>> my_list = [1, 2, 3]
>>> my_tuple = (4, 5, 6)
>>> my_set = {7, 8, 9}
>>> sum(my_list) + sum(my_tuple) + sum(my_set)
45
>>> sum([min(my_list), max(my_tuple), len(my_set)])
10

# round() is particularly useful for formatting data to be printed.
>>> round(3.14159265358979323, 2)
3.14
```

See <https://docs.python.org/3/library/functions.html> for more detailed documentation on all of Python's built-in functions.

Problem 1. Write a function that accepts a list L and returns the minimum, maximum, and average of the entries of L in that order as multiple values (separated by a comma). Can you implement this function in a single line?

Namespaces

Whenever a Python object—a number, data structure, function, or other entity—is created, it is stored somewhere in computer memory. A *name* (or variable) is a reference to a Python object, and a *namespace* is a dictionary that maps names to Python objects.

```
# The number 4 is the object, 'number_of_students' is the name.
>>> number_of_students = 4

# The list is the object, and 'beatles' is the name.
>>> beatles = ["John", "Paul", "George", "Ringo"]

# Python statements defining a function also form an object.
# The name for this function is 'add_numbers'.
>>> def add_numbers(a, b):
...     return a + b
... 
```

A single equals sign assigns a name to an object. If a name is assigned to another name, that new name refers to the same object as the original name.

```
>>> beatles = ["John", "Paul", "George", "Ringo"]
>>> band_members = beatles           # Assign a new name to the list.
>>> print(band_members)
['John', 'Paul', 'George', 'Ringo']
```

To see all of the names in the current namespace, use the built-in function `dir()`. To delete a name from the namespace, use the `del` keyword (**with caution!**).

```
# Add 'stem' to the namespace.
>>> stem = ["Science", "Technology", "Engineering", "Mathematics"]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'stem']

# Remove 'stem' from the namespace.
>>> del stem
>>> "stem" in dir()
False
>>> print(stem)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'stem' is not defined
```

Note

Many programming languages distinguish between variables and pointers. A pointer refers to a variable by storing the address in memory where the corresponding object is stored. Python names are essentially pointers, and traditional pointer operations and cleanup are done automatically. For example, Python automatically deletes objects in memory that have no names assigned to them (no pointers referring to them). This feature is called garbage collection.

Mutability

Every Python object type falls into one of two categories: a *mutable* object, which may be altered at any time, or an *immutable* object, which cannot be altered once created. Attempting to change an immutable object creates a new object in memory. If two names refer to the same mutable object, any changes to the object are reflected in both names since they still both refer to that same object. On the other hand, if two names refer to the same immutable object and one of the values is “changed,” then one name will refer to the original object, and the other will refer to a new object in memory.

Achtung!

Failing to correctly copy mutable objects can cause subtle problems. For example, consider a dictionary that maps items to their base prices. To make a similar dictionary that accounts for a small sales tax, we might try to make a copy by assigning a new name to the first dictionary.

```
>>> holy = {"moly": 1.99, "hand_grenade": 3, "grail": 1975.41}
>>> tax_prices = holy # Try to make a copy for processing.
>>> for item, price in tax_prices.items():
...     # Add a 7 percent tax, rounded to the nearest cent.
...     tax_prices[item] = round(1.07 * price, 2)
...
# Now the base prices have been updated to the total price.
>>> print(tax_prices)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}

# However, dictionaries are mutable, so 'holy' and 'tax_prices' actually
# refer to the same object. The original base prices have been lost.
>>> print(holy)
{'moly': 2.13, 'hand_grenade': 3.21, 'grail': 2113.69}
```

To avoid this problem, explicitly create a copy of the object by casting it as a new structure. Changes made to the copy will not change the original object, since they are distinct objects in memory. To fix the above code, replace the second line with the following:

```
>>> tax_prices = dict(holy)
```


Then, after running the same procedure, the two dictionaries will be different.

Problem 2. Determine which Python object types are mutable and which are immutable by repeating the following experiment for an `int`, `str`, `list`, `tuple`, and `set`.

1. Create an object of the given type and assign a name to it.
2. Assign a new name to the first name.
3. Alter the object via only one of the names (for tuples, use `my_tuple += (1,)`).
4. Check to see if the two names are equal. If they are, then changing one name also changes the other. Thus, both names refer to the same object and the object type is mutable. Otherwise, the names refer to different objects—meaning a new object was created in step 2—and therefore the object type is immutable.

For example, the following experiment shows that `dict` is a mutable type.

```
>>> dict_1 = {1: 'x', 2: 'b'}           # Create a dictionary.
>>> dict_2 = dict_1                     # Assign it a new name.
>>> dict_2[1] = 'a'                     # Change the 'new' dictionary.
>>> dict_1 == dict_2                     # Compare the two names.
True                                    # Both names changed!
```

Print a statement of your conclusions that clearly indicates which object types are mutable and which are immutable.

Achtung!

Mutable objects cannot be put into Python sets or used as keys in Python dictionaries. However, the values of a dictionary may be mutable or immutable.

```
>>> a_dict = {"key": "value"}           # Dictionaries are mutable.
>>> broken = {1, 2, 3, a_dict, a_dict}  # Try putting a dict in a set.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'

>>> okay = {1: 2, "3": a_dict}          # Try using a dict as a value.
```

Modules

A *module* is a Python file containing code that is meant to be used in some other setting, and not necessarily run directly.¹ The `import` statement loads code from a specified Python file. Importing a module containing some functions, classes, or other objects makes those functions, classes, or objects available for use by adding their names to the current namespace.

All import statements should occur at the top of the file, below the header but before any other code. There are several ways to use `import`:

1. `import <module>` makes the specified module available under the alias of its own name.

```
>>> import math                # The name 'math' now gives
>>> math.sqrt(2)               # access to the math module.
1.4142135623730951
```

2. `import <module> as <name>` creates an alias for an imported module. The alias is added to the current namespace, but the module name itself is not.

```
>>> import numpy as np         # The name 'np' gives access to the numpy
>>> np.sqrt(2)                 # module, but the name 'numpy' does not.
1.4142135623730951
>>> numpy.sqrt(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numpy' is not defined
```

3. `from <module> import <object>` loads the specified object into the namespace without loading anything else in the module or the module name itself. This is used most often to access specific functions from a module. The `as` statement can also be tacked on to create an alias.

```
>>> from random import randint # The name 'randint' gives access to the
>>> r = randint(0, 10000)      # randint() function, but the rest of
>>> random.seed(r)             # the random module is unavailable.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'random' is not defined
```

In each case, the final word of the import statement is the name that is added to the namespace.

Running and Importing

Consider the following simple Python module, saved as `example1.py`.

```
# example1.py

data = list(range(4))
```

¹Python files that are primarily meant to be executed, not imported, are often called scripts.

```
def display():
    print("Data:", data)

if __name__ == "__main__":
    display()
    print("This file was executed from the command line or an interpreter.")
else:
    print("This file was imported.")
```

Executing the file from the command line executes the file line by line, including the code under the `if __name__ == "__main__"` clause.

```
$ python example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.
```

Executing the file with IPython's special `%run` command executes each line of the file and also adds the module's names to the current namespace. **This is the quickest way to test individual functions via IPython.**

```
In [1]: %run example1.py
Data: [0, 1, 2, 3]
This file was executed from the command line or an interpreter.

In [2]: display()
Data: [0, 1, 2, 3]
```

Importing the file also executes each line,² but only adds the indicated alias to the namespace. Also, code under the `if __name__ == "__main__"` clause is **not** executed when a file is imported.

```
In [1]: import example1 as ex
This file was imported.

# The module's names are not directly available...
In [2]: display()
-----
NameError                                Traceback (most recent call last)
<ipython-input-2-795648993119> in <module>()
----> 1 display()

NameError: name 'display' is not defined

# ...unless accessed via the module's alias.
In [3]: ex.display()
Data: [0, 1, 2, 3]
```

²Try importing the `this` or `antigravity` modules. Importing these modules actually executes some code.

Problem 3. Create a module called `calculator.py`. Write a function `sum()` that returns the sum of two arguments and a function `product()` that returns the product of two arguments. Also use `import` to add the `sqrt()` function from the `math` module to the namespace. When this file is either run or imported, nothing should be executed.

In your solutions file, import your new custom module. Write a function that accepts two numbers representing the lengths of the sides of a right triangle. Using only the functions from `calculator.py`, calculate and return the length of the hypotenuse of the triangle.

Achtung!

If a module has been imported in IPython and the source code then changes, using `import` again does **not** refresh the name in the IPython namespace. Use `run` instead to correctly refresh the namespace. Consider this example where we test the function `sum_of_squares()`, saved in the file `example2.py`.

```
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x)])
```

In IPython, run the file and test `sum_of_squares()`.

```
# Run the file, adding the function sum_of_squares() to the namespace.
In [1]: %run example2

In [2]: sum_of_squares(3)
Out[2]: 5                                # Should be 14!
```

Since $1^2 + 2^2 + 3^2 = 14$, not 5, something has gone wrong. Modify the source file to correct the mistake, then run the file again in IPython.

```
# example2.py

def sum_of_squares(x):
    """Return the sum of the squares of all positive integers
    less than or equal to x.
    """
    return sum([i**2 for i in range(1,x+1)])    # Include the final term.
```

```
# Run the file again to refresh the namespace.
In [3]: %run example2
```

```
# Now sum_of_squares() is updated to the new, corrected version.
In [4]: sum_of_squares(3)
Out[4]: 14                                # It works!
```

Remember that running or importing a file executes any freestanding code snippets, but any code under an `if __name__ == "__main__":` clause will **only** be executed when the file is run (not when it is imported).

The Python Standard Library

All Python distributions include a collection of modules for accomplishing a variety of common tasks, collectively called the *Python standard library*. Some commonly standard library modules are listed below, and the complete list is at <https://docs.python.org/3/library/>.

Module	Description
<code>cmath</code>	Mathematical functions for complex numbers.
<code>itertools</code>	Tools for iterating through sequences in useful ways.
<code>math</code>	Standard mathematical functions and constants.
<code>random</code>	Random variable generators.
<code>string</code>	Common string literals.
<code>sys</code>	Tools for interacting with the interpreter.
<code>time</code>	Time value generation and manipulation.

Use IPython's object introspection to quickly learn about how to use the various modules and functions in the standard library. Use `?` or `help()` for information on the module or one of its names. To see the entire module's namespace, use the `tab` key.

```
In [1]: import math

In [2]: math?
Type:      module
String form: <module 'math' (built-in)>
Docstring:
This module provides access to the mathematical functions
defined by the C standard.

# Type the module name, a period, then press tab to see the module's namespace.
In [3]: math.  # Press 'tab'.
acos()      cos()      factorial()  isclose()    log2()      tan()
acosh()     cosh()     floor()     isfinite()   modf()      tanh()
asin()      degrees()   fmod()     isinf()     nan         tau
asinh()     e           frexp()    isnan()     pi          trunc()
atan()      erf()      fsum()     ldexp()     pow()
atan2()     erfc()     gamma()    lgamma()    radians()
atanh()     exp()      gcd()      log()       sin()
ceil()      expm1()    hypot()    log10()     sinh()
```

```

    copysign()  fabs()      inf      log1p()    sqrt()

In [3]: math.sqrt?
Signature: math.sqrt(x, /)
Docstring: Return the square root of x.
Type:      builtin_function_or_method

```

The Itertools Module

The `itertools` module makes it easy to iterate over one or more collections in specialized ways.

Function	Description
<code>chain()</code>	Iterate over several iterables in sequence.
<code>cycle()</code>	Iterate over an iterable repeatedly.
<code>combinations()</code>	Return successive combinations of elements in an iterable.
<code>permutations()</code>	Return successive permutations of elements in an iterable.
<code>product()</code>	Iterate over the Cartesian product of several iterables.

```

>>> from itertools import chain, cycle           # Import multiple names.

>>> list(chain("abc", ['d', 'e'], ('f', 'g')))    # Join several
['a', 'b', 'c', 'd', 'e', 'f', 'g']             # sequences together.

>>> for i, number in enumerate(cycle(range(4))):  # Iterate over a single
...     if i > 10:                                # sequence over and over.
...         break
...     print(number, end=' ')
...
0 1 2 3 0 1 2 3 0 1 2

```

A *k-combination* is a set of k elements from a collection where the ordering is unimportant. Thus the combination (a, b) and (b, a) are equivalent because they contain the same elements. On the other hand, a *k-permutation* is a sequence of k elements from a collection where the ordering matters. Even though (a, b) and (b, a) contain the same elements, they are counted as different permutations.

```

>>> from itertools import combinations, permutations

# Get all combinations of length 2 from the iterable "ABC".
>>> list(combinations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'C')]

# Get all permutations of length 2 from "ABC". Note that order matters here.
>>> list(permutations("ABC", 2))
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]

```

Problem 4. The *power set* of a set A , denoted $\mathcal{P}(A)$ or 2^A , is the set of all subsets of A , including the empty set \emptyset and A itself. For example, the power set of the set $A = \{a, b, c\}$ is $2^A = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$.

Write a function that accepts an iterable A . Use an `itertools` function to compute and return the power set of A as a list of sets (why couldn't it be a set of sets in Python?). The empty set should be returned as `set()`.

The Random Module

Many real-life events can be simulated by taking random samples from a probability distribution. For example, a coin flip can be simulated by randomly choosing between the integers 1 (for heads) and 0 (for tails). The `random` module includes functions for sampling from probability distributions and generating random data.

Function	Description
<code>choice()</code>	Choose a random element from a non-empty sequence, such as a list.
<code>randint()</code>	Choose a random integer over a closed interval.
<code>random()</code>	Pick a float from the interval $[0, 1)$.
<code>sample()</code>	Choose several unique random elements from a non-empty sequence.
<code>seed()</code>	Seed the random number generator.
<code>shuffle()</code>	Randomize the ordering of the elements in a list.

Some of the most common `random` utilities involve picking random elements from iterables.

```
>>> import random

>>> numbers = list(range(1,11))      # Get the integers from 1 to 10.
>>> print(numbers)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> random.shuffle(numbers)          # Mix up the ordering of the list.
>>> print(numbers)                   # Note that shuffle() returns nothing.
[5, 9, 1, 3, 8, 4, 10, 6, 2, 7]

>>> random.choice(numbers)           # Pick a single element from the list.
5

>>> random.sample(numbers, 4)        # Pick 4 unique elements from the list.
[5, 8, 3, 2]

>>> random.randint(1,10)             # Pick a random number between 1 and 10.
10
```

The Time Module

The `time` module in the standard library include functions for dealing with time. In particular, the `time()` function measures the number of seconds from a fixed starting point, called “the Epoch” (January 1, 1970 for Unix machines).

```
>>> import time
>>> time.time()
1495243696.645818
```

The `time()` function is useful for measuring how long it takes for code to run: record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed.

```
>>> def time_for_loop(iters):
...     """Time how long it takes to iterate 'iters' times."""
...     start = time.time()           # Clock the starting time.
...     for _ in range(int(iters)):
...         pass
...     end = time.time()              # Clock the ending time.
...     return end - start            # Report the difference.
...
>>> time_for_loop(1e5)                # 1e5 = 100000.
0.005570173263549805
>>> time_for_loop(1e7)                # 1e7 = 10000000.
0.26819777488708496
```

The Sys Module

The `sys` (system) module includes methods for interacting with the Python interpreter. For our purposes, we care about the instance that you call a `.py` file in the command line using ‘python’ followed by the name of a `.py` file or in your `ipython` terminal using ‘%run’ followed by the name of a `.py` file. Remember, to run a `.py` file in the command line, use `cd` to navigate to the folder containing the file before trying to run it. One command we will use is `sys.argv`; which returns a list containing the `.py` file name and all arguments that were passed to the interpreter. This way we can interact with these initial arguments and execute certain parts of our code if the arguments satisfy certain conditions. This will be implemented in problem 5. Note, command line arguments are passed in after the name of the `.py` file and are separated by spaces like so:

```
# the 'python' command followed by a .py file followed by any arguments
$ python yourProgram.py argument1 argument2 argument3

# the equivalent form in an ipython terminal
In[1]: %run yourProgram.py argument1 argument2 argument3
```

Now we can execute code based on the command line arguments by inserting the use of the `sys` module in our `.py` file:


```
>> y = int(input("Enter an integer for y: "))
Enter an integer for y: 16          # Type '16' and press 'enter.'

>>> y
16                                # Note that y contains an integer.
```

Problem 5. *Shut the box* is a popular British pub game that is used to help children learn arithmetic. The player starts with the numbers 1 through 9, and the goal of the game is to eliminate as many of these numbers as possible. At each turn the player rolls two dice, then chooses a set of integers from the remaining numbers that sum up to the sum of the dice roll. These numbers are removed, and the dice are then rolled again. The game ends when none of the remaining integers can be combined to the sum of the dice roll, and the player's final score is the sum of the numbers that could not be eliminated. For a demonstration, see <https://www.youtube.com/watch?v=mwURQC7mjDI>.

Modify your solutions file so that when the file is run with the correct command line arguments (but **not** when it is imported), the user plays a game of shut the box. The provided module `box.py` contains two functions that will be useful in your implementation of the game. You do not need to understand exactly how the functions work, but you do need to be able to import and use them correctly. Their functionality is outlined at the beginning of each function declaration. Your game should match the following specifications:

- Require three total command line arguments: the file name (included by default), the player's name, and a time limit in seconds. If there are not exactly three command line arguments, do not start the game.
- Track the player's remaining numbers, starting with 1 through 9.
- Use the `random` module to simulate rolling two six-sided dice. However, if the sum of the player's remaining numbers is 6 or less, roll only one die.
- The player wins if they have no numbers left, and they lose if they are out of time or if they cannot choose numbers to match the dice roll.
- If the game is not over, print the player's remaining numbers, the sum of the dice roll, and the number of seconds remaining. Prompt the user for numbers to eliminate. The input should be one or more of the remaining integers, separated by spaces. If the user's input is invalid, prompt them for input again before rolling the dice again. (Hint: use `round()` to format the number of seconds remaining nicely.)
- When the game is over, display the player's name, their score, and the total number of seconds since the beginning of the game. Congratulate or mock the player appropriately.

(Hint: **Before you start coding**, write an outline for the entire program, adding one feature at a time. Only start implementing the game after you are completely finished designing it.)

Your game should look similar to the following examples. The characters in red are typed inputs from the user.

```
$ python standard_library.py LuckyDuke 60
```

```
Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Roll: 12
```

```
Seconds left: 60.0
```

```
Numbers to eliminate: 3 9
```

```
Numbers left: [1, 2, 4, 5, 6, 7, 8]
```

```
Roll: 9
```

```
Seconds left: 53.51
```

```
Numbers to eliminate: 8 1
```

```
Numbers left: [2, 4, 5, 6, 7]
```

```
Roll: 7
```

```
Seconds left: 51.39
```

```
Numbers to eliminate: 7
```

```
Numbers left: [2, 4, 5, 6]
```

```
Roll: 2
```

```
Seconds left: 48.24
```

```
Numbers to eliminate: 2
```

```
Numbers left: [4, 5, 6]
```

```
Roll: 11
```

```
Seconds left: 45.16
```

```
Numbers to eliminate: 5 6
```

```
Numbers left: [4]
```

```
Roll: 4
```

```
Seconds left: 42.76
```

```
Numbers to eliminate: 4
```

```
Score for player LuckyDuke: 0 points
```

```
Time played: 15.82 seconds
```

```
Congratulations!! You shut the box!
```

The next two examples show different ways that a player could lose (which they usually do), as well as examples of invalid user input. Use the `box` module's `parse_input()` to detect invalid input.

```
$ python standard_library.py ShakySteve 10
```

```
Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Roll: 7
```

```
Seconds left: 10.0
```

```
Numbers to eliminate: Seven # Must enter a number.
```

```
Invalid input

Seconds left: 7.64
Numbers to eliminate: 1, 2, 4           # Do not use commas.
Invalid input

Seconds left: 4.55
Numbers to eliminate: 1 2 3           # Numbers don't sum to the roll.
Invalid input

Seconds left: 2.4
Numbers to eliminate: 1 2 4

Numbers left: [3, 5, 6, 7, 8, 9]
Roll: 8
Seconds left: 0.31
Numbers to eliminate: 8
Game over!                             # Time is up!

Score for player ShakySteve: 30 points
Time played: 11.77 seconds
Better luck next time >:)
```

```
$ python standard_library.py SnakeEyesTom 10000

Numbers left: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Seconds left: 10000.0
Numbers to eliminate: 2

Numbers left: [1, 3, 4, 5, 6, 7, 8, 9]
Roll: 2
Game over!                             # Numbers cannot match roll.

Score for player SnakeEyesTom: 43 points
Time played: 1.53 seconds
Better luck next time >:)
```

Additional Material

More Built-in Functions

The following built-in functions are worth knowing, especially for working with iterables and writing very readable conditional statements.

Function	Description
<code>all()</code>	Return <code>True</code> if <code>bool(entry)</code> evaluates to <code>True</code> for every entry in the input iterable.
<code>any()</code>	Return <code>True</code> if <code>bool(entry)</code> evaluates to <code>True</code> for any entry in the input iterable.
<code>bool()</code>	Evaluate a single input object as <code>True</code> or <code>False</code> .
<code>eval()</code>	Execute a string as Python code and return the output.
<code>map()</code>	Apply a function to every item of the input iterable and return an iterable of the results.

```
>>> from random import randint
# Get 5 random numbers between 1 and 10, inclusive.
>>> numbers = [randint(1,10) for _ in range(5)]

# If all of the numbers are less than 8, print the list.
>>> if all([num < 8 for num in numbers]):
...     print(numbers)
...
[1, 5, 6, 3, 3]

# If none of the numbers are divisible by 3, print the list.
>>> if not any([num % 3 == 0 for num in numbers]):
...     print(numbers)
...
```

Two-Player Shut the Box

Consider modifying your shut the box program so that it pits two players against each other (one player tries to shut the box while the other tries to keep it open). The first player plays a regular round as described in Problem 5. Suppose he or she eliminates every number but 2, 3, and 6. The second player then begins a round with the numbers 1, 4, 5, 7, 8, and 9, the numbers that the first player had eliminated. If the second player loses, the first player gets another round to try to shut the box with the numbers that the second player had eliminated. Play continues until one of the players eliminates their entire list. In addition, each player should have their own time limit that only ticks down during their turn. If time runs out on your turn, you lose no matter what.

Python Packages

Large programming projects often have code spread throughout several folders and files. In order to get related files in different folders to communicate properly, the associated directories must

be organized into a Python *packages*. This is a common procedure when creating smart phone applications and other programs that have graphical user interfaces (GUIs).

A package is simply a folder that contains a file called `__init__.py`. This file is always executed first whenever the package is used. A package must also have a file called `__main__.py` in order to be executable. Executing the package will run `__init__.py` and then `__main__.py`, but importing the package will only run `__init__.py`.

Use the regular syntax to import a module or subpackage that is in the current package, and use `from <subpackage.module> import <object>` to load a module within a subpackage. Once a name has been loaded into a package's `__init__.py`, other files in the same package can load the same name with `from . import <object>`. To access code in the directory one level above the current directory, use the syntax `from .. import <object>`. This tells the interpreter to go up one level and import the object from there. This is called an *explicit relative import* and cannot be done in files that are executed directly (like `__main__.py`).

Finally, to execute a package, run Python from the shell with the flag `-m` (for “module-name”) and exclude the extension `.py`.

```
$ python -m package_name
```

See <https://docs.python.org/3/tutorial/modules.html#packages> for examples and more details.

4

Introduction to NumPy

Lab Objective: *NumPy is a powerful Python package for manipulating data with multi-dimensional vectors. Its versatility and speed makes Python an ideal language for applied and computational mathematics. In this lab we introduce basic NumPy data structures and operations as a first step to numerical computing in Python.*

Arrays

In many algorithms, data can be represented mathematically as a *vector* or a *matrix*. Conceptually, a vector is just a list of numbers and a matrix is a two-dimensional list of numbers (a list of lists). However, even basic linear algebra operations like matrix multiplication are cumbersome to implement and slow to execute when data is stored this way. The *NumPy* module¹ [?, ?, ?] offers a much better solution.

The basic object in NumPy is the *array*, which is conceptually similar to a matrix. The NumPy array class is called `ndarray` (for “ n -dimensional array”). The simplest way to explicitly create a 1-D `ndarray` is to define a list, then cast that list as an `ndarray` with NumPy’s `array()` function.

```
>>> import numpy as np

# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])

# The string representation has no commas or an array() label.
>>> print(np.array([1, 3, 5, 7, 9]))
[1 3 5 7 9]
```

The alias “`np`” is standard in the Python community.

An `ndarray` can have arbitrarily many dimensions. A 2-D array is a 1-D array of 1-D arrays (like a list of lists), a 3-D array is a 1-D array of 2-D arrays (a list of lists of lists), and, more generally, an n -dimensional array is a 1-D array of $(n - 1)$ -dimensional arrays (a list of lists of lists of lists...). Each dimension is called an *axis*. For a 2-D array, the 0-axis indexes the rows and the 1-axis indexes the columns. Elements are accessed using brackets and indices, with the axes separated by commas.

¹NumPy is not part of the standard library, but it is included in most Python distributions.

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array( [ [1, 2, 3],[4, 5, 6] ] )
>>> print(A)
[[1 2 3]
 [4 5 6]]

# Access elements of the array with brackets.
>>> print(A[0, 1], A[1, 2])
2 6

# The elements of a 2-D array are 1-D arrays.
>>> A[0]
array([1, 2, 3])
```

Problem 1. There are two main ways to perform matrix multiplication in NumPy: with NumPy's `dot()` function (`np.dot(A, B)`), or with the `@` operator (`A @ B`). Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 3 & -1 & 4 \\ 1 & 5 & -9 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 6 & -5 & 3 \\ 5 & -8 & 9 & 7 \\ 9 & -3 & -2 & -3 \end{bmatrix}$$

Return the matrix product AB .

For examples of array initialization and matrix multiplication, use object introspection in IPython to look up the documentation for `np.ndarray`, `np.array()` and `np.dot()`.

```
In [1]: import numpy as np

In [2]: np.array?          # press 'enter'
```

Achtung!

The `@` operator was not introduced until Python 3.5. It triggers the `__matmul__()` magic method,^a which for the `ndarray` is essentially a wrapper around `np.dot()`. If you are using a previous version of Python, always use `np.dot()` to perform basic matrix multiplication.

^aSee the lab on Object Oriented Programming for an overview of magic methods.

Basic Array Operations

NumPy arrays behave differently with respect to the binary arithmetic operators `+` and `*` than Python lists do. For lists, `+` concatenates two lists and `*` replicates a list by a scalar amount (strings also behave this way).


```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

# Mutlipleication concatenates a list with itself a given number of times.
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

NumPy arrays act like mathematical vectors and matrices: `+` and `*` perform component-wise addition or multiplication.

```
>>> x, y = np.array([1, 2, 3]), np.array([4, 5, 6])

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10                                # Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4                                # Multiply each entry of x by 4.
array([ 4,  8, 12])

# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])

# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

Problem 2. Write a function that defines the following matrix as a NumPy array.

$$A = \begin{bmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ -5 & 3 & 1 \end{bmatrix}$$

Return the matrix $-A^3 + 9A^2 - 15A$.

In this context, $A^2 = AA$ (the matrix product, not the component-wise square). The somewhat surprising result is a demonstration of the Cayley-Hamilton theorem.

Array Attributes

An `ndarray` object has several attributes, some of which are listed below.

Attribute	Description
<code>dtype</code>	The type of the elements in the array.
<code>ndim</code>	The number of axes (dimensions) of the array.
<code>shape</code>	A tuple of integers indicating the size in each dimension.
<code>size</code>	The total number of elements in the array.

```
>>> A = np.array([[1, 2, 3],[4, 5, 6]])

# 'A' is a 2-D array with 2 rows, 3 columns, and 6 entries.
>>> print(A.ndim, A.shape, A.size)
2 (2, 3) 6
```

Note that `ndim` is the number of entries in `shape`, and that the `size` of the array is the product of the entries of `shape`.

Array Creation Routines

In addition to casting other structures as arrays via `np.array()`, NumPy provides efficient ways to create certain commonly-used arrays.

Function	Returns
<code>arange()</code>	Array of sequential integers (like <code>list(range())</code>).
<code>eye()</code>	2-D array with ones on the diagonal and zeros elsewhere.
<code>ones()</code>	Array of given shape and type, filled with ones.
<code>ones_like()</code>	Array of ones with the same shape and type as a given array.
<code>zeros()</code>	Array of given shape and type, filled with zeros.
<code>zeros_like()</code>	Array of zeros with the same shape and type as a given array.
<code>full()</code>	Array of given shape and type, filled with a specified value.
<code>full_like()</code>	Full array with the same shape and type as a given array.

Each of these functions accepts the keyword argument `dtype` to specify the data type. Common types include `np.bool_`, `np.int64`, `np.float64`, and `np.complex128`.

```
# A 1-D array of 5 zeros.
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# A 2x5 matrix (2-D array) of integer ones.
>>> np.ones((2,5), dtype=np.int) # The shape is specified as a tuple.
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

# The 2x2 identity matrix.
>>> I = np.eye(2)
>>> print(I)
[[ 1.  0.]
 [ 0.  1.]]

# Array of 3s the same size as 'I'.
>>> np.full_like(I, 3) # Equivalent to np.full(I.shape, 3).
array([[ 3.,  3.],
       [ 3.,  3.]])
```

Unlike native Python data structures, **all elements of a NumPy array must be of the same data type**. To change an existing array's data type, use the array's `astype()` method.

```
# A list of integers becomes an array of integers.
>>> x = np.array([0, 1, 2, 3, 4])
>>> print(x)
[0 1 2 3 4]
>>> x.dtype
dtype('int64')

# Change the data type to one of NumPy's float types.
>>> x = x.astype(np.float64)      # Equivalent to x = np.float64(x).
>>> print(x)
[ 0.  1.  2.  3.  4.]           # Floats are displayed with periods.
>>> x.dtype
dtype('float64')
```

The following functions are for dealing with the diagonal, upper, or lower portion of an array.

Function	Description
<code>diag()</code>	Extract a diagonal or construct a diagonal array.
<code>tril()</code>	Get the lower-triangular portion of an array by replacing entries above the diagonal with zeros.
<code>triu()</code>	Get the upper-triangular portion of an array by replacing entries below the diagonal with zeros.

```
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Get only the upper triangular entries of 'A'.
>>> np.triu(A)
array([[1, 2, 3],
       [0, 5, 6],
       [0, 0, 9]])

# Get the diagonal entries of 'A' as a 1-D array.
>>> np.diag(A)
array([1, 5, 9])

# diag() can also be used to create a diagonal matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[ 1,  0,  0],
       [ 0, 11,  0],
       [ 0,  0, 111]])
```

See <http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html> for the official documentation on NumPy's array creation routines.

Problem 3. Write a function that defines the following matrices as NumPy arrays using the functions presented in this section (not `np.array()`). Calculate the matrix product ABA . Change the data type of the resulting matrix to `np.int64`, then return it.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} -1 & 5 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & 5 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & 5 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & 5 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & 5 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Data Access

Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access elements of the array with slicing syntax.
>>> x[3]                                # The element at index 3.
3
>>> x[:3]                                # Everything up to index 3 (exclusive).
array([0, 1, 2])
>>> x[3:]                                # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                              # The elements from index 3 to 8.
array([3, 4, 5, 6, 7])

>>> A = np.array([[0,1,2,3,4],[5,6,7,8,9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> A[1, 2]                              # The element at row 1, column 2.
7
>>> A[:, 2:]                             # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

Note

Indexing and slicing operations return a view of the array. Changing a view of an array also changes the original array. In other words, **arrays are mutable**. To create a copy of an array, use `np.copy()` or the array's `copy()` method. Changes to a copy of an array does not affect the original array, but copying an array uses more time and memory than getting a view.

Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *mask*) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)      # The integers from 0 to 50 by tens.
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])   # Get the 3rd, 1st, and 4th elements.
>>> x[index]                      # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]                      # Get the 0th and 3rd entries.
array([ 0, 30])
```

Fancy indexing is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like `<` and `==` to create masks.

```
>>> y = np.arange(10, 20, 2)      # Every other integers from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False,  True,  True], dtype=bool)
>>> y[mask]                      # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14 100 100]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

Problem 4. Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the resulting array.

Array Manipulation

Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> A = np.arange(12)                # The integers from 0 to 12 (exclusive).
>>> print(A)
[ 0  1  2  3  4  5  6  7  8  9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3,4))                 # The new shape is specified as a tuple.
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2,-1))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> A = np.arange(12).reshape((3,4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                      # Equivalent to A.reshape(A.size)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                              # Equivalent to np.transpose(A).
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

Note

By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into “flat” 1-D arrays. For example, by default an array will have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented to purposefully work well with 1-D “row arrays”.

```
>>> A = np.arange(10).reshape((2,5))
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])

# Slicing out a column of A still produces a "flat" 1-D array.
>>> x = A[:,1]                # All of the rows, column 1.
>>> x
array([1, 6])                # Not array([[1],
                             #           [6]])
>>> x.shape
(2,)
>>> x.ndim
1
```

However, it is occasionally necessary to change a 1-D array into a “column array”. Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((-1,1))        # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
       [1],
       [2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

Function	Description
<code>concatenate()</code>	Join a sequence of arrays along an existing axis
<code>hstack()</code>	Stack arrays in sequence horizontally (column wise).
<code>vstack()</code>	Stack arrays in sequence vertically (row wise).
<code>column_stack()</code>	Stack 1-D arrays as columns into a 2-D array.

```

>>> A = np.arange(6).reshape((2,3))
>>> B = np.zeros((4,3))

# vstack() stacks arrays vertically (row-wise).
>>> np.vstack((A,B,A))
array([[ 0.,  1.,  2.],           # A
       [ 3.,  4.,  5.],
       [ 0.,  0.,  0.],           # B
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  2.],           # A
       [ 3.,  4.,  5.]])

>>> A = A.T
>>> B = np.ones((3,4))

# hstack() stacks arrays horizontally (column-wise).
>>> np.hstack((A,B,A))
array([[ 0.,  3.,  1.,  1.,  1.,  1.,  0.,  3.],
       [ 1.,  4.,  1.,  1.,  1.,  1.,  1.,  4.],
       [ 2.,  5.,  1.,  1.,  1.,  1.,  2.,  5.]])

# column_stack() stacks arrays horizontally, including 1-D arrays.
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[ 0.,  3.,  0.,  1.,  2.],
       [ 1.,  4.,  0.,  1.,  2.],
       [ 2.,  5.,  0.,  1.,  2.]])

```

See <http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html> for more array manipulation routines and documentation.

Problem 5. Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 0 & 0 \\ 3 & 3 & 0 \\ 3 & 3 & 3 \end{bmatrix} \quad C = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

Use NumPy's stacking functions to create and return the block matrix:

$$\begin{bmatrix} \mathbf{0} & A^T & I \\ A & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & C \end{bmatrix},$$

where I is the 3×3 identity matrix and each $\mathbf{0}$ is a matrix of all zeros of appropriate size.

A block matrix of this form is used in the interior point method for linear optimization.

Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape, such as element-wise addition. *Array broadcasting* extends such operations to accept some (but not all) operands with different shapes, and occurs automatically whenever possible.

Suppose, for example, that we would like to add different values to the columns of an $m \times n$ matrix A . Adding a 1-D array x with the n entries to A will automatically do this correctly. To add different values to the different rows of A , first reshape a 1-D array of m values into a column array. Broadcasting then correctly takes care of the operation.

Broadcasting can also occur between two 1-D arrays, once they are reshaped appropriately.

```
>>> A = np.arange(12).reshape((4,3))
>>> x = np.arange(3)
>>> A
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> x
array([0, 1, 2])

# Add the entries of 'x' to the corresponding columns of 'A'.
>>> A + x
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])

>>> y = np.arange(0, 40, 10).reshape((4,1))
>>> y
array([[ 0],
       [10],
       [20],
       [30]])

# Add the entries of 'y' to the corresponding rows of 'A'.
>>> A + y
array([[ 0,  1,  2],
       [13, 14, 15],
       [26, 27, 28],
       [39, 40, 41]])

# Add 'x' and 'y' together with array broadcasting.
>>> x + y
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Numerical Computing with NumPy

Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an array.

Function	Description
<code>abs()</code> or <code>absolute()</code>	Calculate the absolute value element-wise.
<code>exp()</code> / <code>log()</code>	Exponential (e^x) / natural log element-wise.
<code>maximum()</code> / <code>minimum()</code>	Element-wise maximum / minimum of two arrays.
<code>sqrt()</code>	The positive square-root, element-wise.
<code>sin()</code> , <code>cos()</code> , <code>tan()</code> , etc.	Element-wise trigonometric operations.

```
>>> x = np.arange(-2,3)
>>> print(x, np.abs(x))           # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                    # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.          ,  0.84147098,  0.90929743])
```

See <http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs> for a more comprehensive list of universal functions.

Achtung!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.0855369232

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.          ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

Method	Returns
<code>all()</code>	<code>True</code> if all elements evaluate to <code>True</code> .
<code>any()</code>	<code>True</code> if any elements evaluate to <code>True</code> .
<code>argmax()</code>	Index of the maximum value.
<code>argmin()</code>	Index of the minimum value.
<code>argsort()</code>	Indices that would sort the array.
<code>clip()</code>	restrict values in an array to fit within a given range
<code>max()</code>	The maximum element of the array.
<code>mean()</code>	The average value of the array.
<code>min()</code>	The minimum element of the array.
<code>sort()</code>	Return nothing; sort the array in-place.
<code>std()</code>	The standard deviation of the array.
<code>sum()</code>	The sum of the elements of the array.
<code>var()</code>	The variance of the array.

Each of these `np.ndarray` methods has an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed can operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an n -D array, the return value is an $(n - 1)$ -D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar. Refer to the NumPy Visual Guide in the appendix for more visual examples.

```
>>> A = np.arange(9).reshape((3,3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
>>> A.min(axis=0)           # np.array([min(A[:,i]) for i in range(3)])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)          # np.array([sum(A[i,:]) for i in range(3)])
array([3, 12, 21])
```

See <http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html> for a more comprehensive list of array methods.

Problem 6. A matrix is called *row-stochastic*^a if its rows each sum to 1. Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function that accepts a matrix (as a 2-D NumPy array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

^aSimilarly, a matrix is called column-stochastic if its columns each sum to 1.

Problem 7. This problem comes from <https://projecteuler.net>.

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

```

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

```

The product of these numbers is $26 \times 63 \times 78 \times 14 = 1788696$. Write a function that returns the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the grid.

For convenience, this array has been saved in the file `grid.npy`. Use the following syntax to extract the array:

```
>>> grid = np.load("grid.npy")
```

One way to approach this problem is to iterate through the rows and columns of the array, checking small slices of the array at each iteration and updating the current largest product. Array slicing, however, provides a much more efficient solution.

The naïve method for computing the greatest product of four adjacent numbers in a horizontal row might be as follows:

```
>>> winner = 0
>>> for i in range(20):
...     for j in range(17):
...         winner = max(np.prod(grid[i,j:j+4]), winner)
...
>>> winner
48477312
```

Instead, use array slicing to construct a single array where the (i, j) th entry is the product of the four numbers to the right of the (i, j) th entry in the original grid. Then find the largest element in the new array.

```
>>> np.max(grid[:, :-3] * grid[:, 1:-2] * grid[:, 2:-1] * grid[:, 3:])
48477312
```

Use slicing to similarly find the greatest products of four vertical, right diagonal, and left diagonal adjacent numbers.

(Hint: Consider drawing the portions of the grid that each slice in the above code covers, like the examples in the visual guide. Then draw the slices that produce vertical, right diagonal, or left diagonal sequences, and translate the pictures into slicing syntax.)

Achtung!

All of the examples in this lab use NumPy arrays, objects of type `np.ndarray`. NumPy also has a “matrix” data structure called `np.matrix` that was built specifically for MATLAB users who are transitioning to Python and NumPy. It behaves slightly differently than the regular array class, and can cause some unexpected and subtle problems.

For consistency (and your sanity), **never** use a NumPy matrix; **always** use NumPy arrays. If necessary, cast a matrix object as an array with `np.array()`.

Additional Material

Random Sampling

The submodule `np.random` holds many functions for creating arrays of random values chosen from probability distributions such as the uniform, normal, and multinomial distributions. It also contains some utility functions for getting non-distributional random samples, such as random integers or random samples from a given array.

Function	Description
<code>choice()</code>	Take random samples from a 1-D array.
<code>random()</code>	Uniformly distributed floats over $[0, 1)$.
<code>randint()</code>	Random integers over a half-open interval.
<code>random_integers()</code>	Random integers over a closed interval.
<code>randn()</code>	Sample from the standard normal distribution.
<code>permutation()</code>	Randomly permute a sequence / generate a random sequence.
Function	Distribution
<code>beta()</code>	Beta distribution over $[0, 1]$.
<code>binomial()</code>	Binomial distribution.
<code>exponential()</code>	Exponential distribution.
<code>gamma()</code>	Gamma distribution.
<code>geometric()</code>	Geometric distribution.
<code>multinomial()</code>	Multivariate generalization of the binomial distribution.
<code>multivariate_normal()</code>	Multivariate generalization of the normal distribution.
<code>normal()</code>	Normal / Gaussian distribution.
<code>poisson()</code>	Poisson distribution.
<code>uniform()</code>	Uniform distribution.

Note that many of these functions have counterparts in the standard library's `random` module. These NumPy functions, however, are much better suited for working with large collections of random samples.

```
# 5 uniformly distributed values in the interval [0, 1).
>>> np.random.random(5)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])

# A 2x5 matrix (2-D array) of integers in the interval [10, 20).
>>> np.random.randint(10, 20, (2,5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

Saving and Loading Arrays

It is often useful to save an array as a file for later use. NumPy provides several easy methods for saving and loading array data.

Function	Description
<code>save()</code>	Save a single array to a <code>.npy</code> file.
<code>savez()</code>	Save multiple arrays to a <code>.npz</code> file.
<code>savetxt()</code>	Save a single array to a <code>.txt</code> file.
<code>load()</code>	Load and return an array or arrays from a <code>.npy</code> or <code>.npz</code> file.
<code>loadtxt()</code>	Load and return an array from a text file.

```
# Save a 100x100 matrix of uniformly distributed random values.
>>> x = np.random.random((100,100))
>>> np.save("uniform.npy", x)          # Or np.savetxt("uniform.txt", x).

# Read the array from the file and check that it matches the original.
>>> y = np.load("uniform.npy")         # Or np.loadtxt("uniform.txt").
>>> np.allclose(x, y)                  # Check that x and y are close entry-wise.
True
```

To save several arrays to a single file, specify a keyword argument for each array in `np.savez()`. Then `np.load()` will return a dictionary-like object with the keyword parameter names from the save command as the keys.

```
# Save two 100x100 matrices of normally distributed random values.
>>> x = np.random.randn(100,100)
>>> y = np.random.randn(100,100)
>>> np.savez("normal.npz", first=x, second=y)

# Read the arrays from the file and check that they match the original.
>>> arrays = np.load("normal.npz")
>>> np.allclose(x, arrays["first"])
True
>>> np.allclose(y, arrays["second"])
True
```


5

Object-oriented Programming

Lab Objective: *Python is a class-based language. A class is a blueprint for an object that binds together specified variables and routines. Creating and using custom classes is often a good way to write clean, efficient, well-designed programs. In this lab we learn how to define and use Python classes. In subsequent labs we will often create customized classes for use in algorithms.*

Classes

A Python *class* is a code block that defines a custom object and determines its behavior. The `class` key word defines and names a new class. Other statements follow, indented below the class name, to determine the behavior of objects instantiated by the class.

A class needs a method called a *constructor* that is called whenever the class instantiates a new object. The constructor specifies the initial state of the object. In Python, a class's constructor is always named `__init__()`. For example, the following code defines a class for storing information about backpacks.

```
class Backpack:
    """A Backpack object class. Has a name and a list of contents.

    Attributes:
        name (str): the name of the backpack's owner.
        contents (list): the contents of the backpack.
    """
    def __init__(self, name):          # This function is the constructor.
        """Set the name and initialize an empty list of contents.

        Parameters:
            name (str): the name of the backpack's owner.
        """
        self.name = name               # Initialize some attributes.
        self.contents = []
```

An *attribute* is a variable stored within an object. The `Backpack` class has two attributes: `name` and `contents`. In the body of the class definition, attributes are assigned and accessed via the name `self`. This name refers to the object internally once it has been created.

Instantiation

The `class` code block above only defines a blueprint for backpack objects. To create an actual backpack object, call the class name like a function. This triggers the constructor and returns a new *instance* of the class, an object whose type is the class.

```
# Import the Backpack class and instantiate an object called 'my_backpack'.
>>> from object_oriented import Backpack
>>> my_backpack = Backpack("Fred")
>>> type(my_backpack)
<class 'object_oriented.Backpack'>

# Access the object's attributes with a period and the attribute name.
>>> print(my_backpack.name, my_backpack.contents)
Fred []

# The object's attributes can be modified after instantiation.
>>> my_backpack.name = "George"
>>> print(my_backpack.name, my_backpack.contents)
George []
```

Note

Every object in Python has some built-in attributes. For example, modules have a `__name__` attribute that identifies the scope in which it is being executed. If the module is being run directly, and not imported, then `__name__` is set to `"__main__"`. Therefore, any commands under an `if __name__ == "__main__":` clause are ignored when the module is imported.

Methods

In addition to storing variables as attributes, classes can have functions attached to them. A function that belongs to a specific class is called a *method*.

```
class Backpack:
    # ...
    def put(self, item):
        """Add an item to the backpack's list of contents."""
        self.contents.append(item) # Use 'self.contents', not just 'contents'.

    def take(self, item):
        """Remove an item from the backpack's list of contents."""
        self.contents.remove(item)
```

The first argument of each method must be `self`, to give the method access to the attributes and other methods of the class. The `self` argument is only included in the declaration of the class methods, **not** when calling the methods on an instantiation of the class.

```
# Add some items to the backpack object.
>>> my_backpack.put("notebook")      # my_backpack is passed implicitly to
>>> my_backpack.put("pencils")       # Backpack.put() as the first argument.
>>> my_backpack.contents
['notebook', 'pencils']

# Remove an item from the backpack.    # This is equivalent to
>>> my_backpack.take("pencils")      # Backpack.take(my_backpack, "pencils")
>>> my_backpack.contents
['notebook']
```

Problem 1. Expand the Backpack class to match the following specifications.

1. Modify the constructor so that it accepts three total arguments: `name`, `color`, and `max_size` (in that order). Make `max_size` a keyword argument that defaults to 5. Store each input as an attribute.
2. Modify the `put()` method to check that the backpack does not go over capacity. If there are already `max_size` items or more, print “No Room!” and do not add the item to the contents list.
3. Write a new method called `dump()` that resets the contents of the backpack to an empty list. This method should not receive any arguments (except `self`).
4. Documentation is especially important in classes so that the user knows what an object’s attributes represent and how to use methods appropriately. Update (or write) the docstrings for the `__init__()`, `put()`, and `dump()` methods, as well as the actual class docstring (under `class` but before `__init__()`) to reflect the changes from parts 1-3 of this problem.

To ensure that your class works properly, write a test function outside of the Backpack class that instantiates and analyzes a Backpack object.

```
def test_backpack():
    testpack = Backpack("Barry", "black")      # Instantiate the object.
    if testpack.name != "Barry":              # Test an attribute.
        print("Backpack.name assigned incorrectly")
    for item in ["pencil", "pen", "paper", "computer"]:
        testpack.put(item)                    # Test a method.
    print("Contents:", testpack.contents)
    # ...
```

Inheritance

To create a new class that is similar to one that already exists, it is often better to *inherit* the methods and attributes from an existing class rather than create a new class from scratch. This creates a *class hierarchy*: a class that inherits from another class is called a *subclass*, and the class that a subclass inherits from is called a *superclass*. To define a subclass, add the name of the superclass as an argument at the end of the `class` declaration.

For example, since a knapsack is a kind of backpack (but not all backpacks are knapsacks), we create a special `Knapsack` subclass that inherits the structure and behaviors of the `Backpack` class and adds some extra functionality.

```
# Inherit from the Backpack class in the class definition.
class Knapsack(Backpack):
    """A Knapsack object class. Inherits from the Backpack class.
    A knapsack is smaller than a backpack and can be tied closed.

    Attributes:
        name (str): the name of the knapsack's owner.
        color (str): the color of the knapsack.
        max_size (int): the maximum number of items that can fit inside.
        contents (list): the contents of the backpack.
        closed (bool): whether or not the knapsack is tied shut.
    """
    def __init__(self, name, color, max_size=3):
        """Use the Backpack constructor to initialize the name, color,
        and max_size attributes. A knapsack only holds 3 item by default.

        Parameters:
            name (str): the name of the knapsack's owner.
            color (str): the color of the knapsack.
            max_size (int): the maximum number of items that can fit inside.
        """
        Backpack.__init__(self, name, color, max_size)
        self.closed = True
```

A subclass may have new attributes and methods that are unavailable to the superclass, such as the `closed` attribute in the `Knapsack` class. If methods from the superclass need to be changed for the subclass, they can be overridden by defining them again in the subclass. New methods can be included normally.

```
class Knapsack(Backpack):
    # ...
    def put(self, item):
        # Override the put() method.
        """If the knapsack is untied, use the Backpack.put() method."""
        if self.closed:
            print("I'm closed!")
        else:
            # Use Backpack's original put().
            Backpack.put(self, item)
```

```

def take(self, item):          # Override the take() method.
    """If the knapsack is untied, use the Backpack.take() method."""
    if self.closed:
        print("I'm closed!")
    else:
        Backpack.take(self, item)

def weight(self):              # Define a new method just for knapsacks.
    """Calculate the weight of the knapsack by counting the length of the
    string representations of each item in the contents list.
    """
    return sum(len(str(item)) for item in self.contents)

```

Since `Knapsack` inherits from `Backpack`, a knapsack object **is** a backpack object. All methods defined in the `Backpack` class are available as instances of the `Knapsack` class. For example, the `dump()` method is available even though it is not defined explicitly in the `Knapsack` class.

The built-in function `issubclass()` shows whether or not one class is derived from another. Similarly, `isinstance()` indicates whether or not an object belongs to a specified class hierarchy. Finally, `hasattr()` shows whether or not a class or object **has** a specified attribute or method.

```

>>> from object_oriented import Knapsack
>>> my_knapsack = Knapsack("Brady", "brown")

# A Knapsack is a Backpack, but a Backpack is not a Knapsack.
>>> print(issubclass(Knapsack, Backpack), issubclass(Backpack, Knapsack))
True False
>>> isinstance(my_knapsack, Knapsack) and isinstance(my_knapsack, Backpack)
True

# The put() and take() method now require the knapsack to be open.
>>> my_knapsack.put('compass')
I'm closed!

# Open the knapsack and put in some items.
>>> my_knapsack.closed = False
>>> my_knapsack.put("compass")
>>> my_knapsack.put("pocket knife")
>>> my_knapsack.contents
['compass', 'pocket knife']

# The Knapsack class has a weight() method, but the Backpack class does not.
>>> print(hasattr(my_knapsack, 'weight'), hasattr(my_backpack, 'weight'))
True False

# The dump method is inherited from the Backpack class.
>>> my_knapsack.dump()
>>> my_knapsack.contents
[]

```

Problem 2. Write a `Jetpack` class that inherits from the `Backpack` class.

1. Override the constructor so that in addition to a name, color, and maximum size, it also accepts an amount of fuel. Change the default value of `max_size` to 2, and set the default value of fuel to 10. Store the fuel as an attribute.
2. Add a `fly()` method that accepts an amount of fuel to be burned and decrements the fuel attribute by that amount. If the user tries to burn more fuel than remains, print “Not enough fuel!” and do not decrement the fuel.
3. Override the `dump()` method so that both the contents and the fuel tank are emptied.
4. Write clear, detailed docstrings for the class and each of its methods.

Note

All classes are subclasses of the built-in `object` class, even if no parent class is specified in the class definition. In fact, the syntax “`class ClassName(object):`” is not uncommon (or incorrect) for the class declaration, and is equivalent to the simpler “`class ClassName:`”.

Magic Methods

A *magic method* is a special method used to make an object behave like a built-in data type. Magic methods begin and end with two underscores, like the constructor `__init__()`. Every Python object is automatically endowed with several magic methods, which can be revealed through IPython.

```
In [1]: %run object_oriented.py

In [2]: b = Backpack("Oscar", "green")

In [3]: b.          # Press 'tab' to see standard methods and attributes.
        color      max_size take()
        contents   name
        dump()     put()

In [3]: b.__        # Press 'tab' to see magic methods and hidden attributes.
        __add__()   __getattr__   __new__()
        __class__   __gt__        __reduce__()
        __delattr__ __hash__      __reduce_ex__()
        __dict__    __init__()    __repr__
        __dir__()   __init_subclass__ __setattr__
        __doc__     __le__        __sizeof__()
        __eq__      __lt__        __str__
        __format__() __module__   __subclasshook__()
        __ge__      __ne__        __weakref__
```

Note

Many programming languages distinguish between public and private variables. In Python, all attributes are public, period. However, attributes that start with an underscore are hidden from the user, which is why magic methods do not show up at first in the preceding code box.

The more common magic methods define how an object behaves with respect to addition and other binary operations. For example, how should addition be defined for backpacks? A simple option is to add the number of contents. Then if backpack A has 3 items and backpack B has 5 items, `A + B` should return 8. To incorporate this idea, we implement the `__add__()` magic method.

```
class Backpack:
    # ...
    def __add__(self, other):
        """Add the number of contents of each Backpack."""
        return len(self.contents) + len(other.contents)
```

Using the `+` binary operator on two `Backpack` objects calls the class's `__add__()` method. The object on the left side of the `+` is passed in to `__add__()` as `self` and the object on the right side of the `+` is passed in as `other`.

```
>>> pack1 = Backpack("Rose", "red")
>>> pack2 = Backpack("Carly", "cyan")

# Put some items in the backpacks.
>>> pack1.put("textbook")
>>> pack2.put("water bottle")
>>> pack2.put("snacks")

# Add the backpacks together.
>>> pack1 + pack2                                # Equivalent to pack1.__add__(pack2).
3
```

Comparisons

Magic methods also facilitate object comparisons. For example, the `__lt__()` method corresponds to the `<` operator. Suppose one backpack is considered “less” than another if it has fewer items in its list of contents.

```
class Backpack(object)
    # ...
    def __lt__(self, other):
        """If 'self' has fewer contents than 'other', return True.
        Otherwise, return False.
        """
        return len(self.contents) < len(other.contents)
```

Using the `<` binary operator on two `Backpack` objects calls `__lt__()`. As with addition, the object on the left side of the `<` operator is passed to `__lt__()` as `self`, and the object on the right is passed in as `other`.

```
>>> pack1, pack2 = Backpack("Maggy", "magenta"), Backpack("Yolanda", "yellow")
>>> pack1 < pack2
False
# Equivalent to pack1.__lt__(pack2).

>>> pack2.put('pencils')
>>> pack1 < pack2
True
```

Comparison methods should return either `True` or `False`, while methods like `__add__()` might return a numerical value or another kind of object.

Method	Arithmetic Operator	Method	Comparison Operator
<code>__add__()</code>	<code>+</code>	<code>__lt__()</code>	<code><</code>
<code>__sub__()</code>	<code>-</code>	<code>__le__()</code>	<code><=</code>
<code>__mul__()</code>	<code>*</code>	<code>__gt__()</code>	<code>></code>
<code>__pow__()</code>	<code>**</code>	<code>__ge__()</code>	<code>>=</code>
<code>__truediv__()</code>	<code>/</code>	<code>__eq__()</code>	<code>==</code>
<code>__floordiv__()</code>	<code>//</code>	<code>__ne__()</code>	<code>!=</code>

Table 5.1: Common magic methods for arithmetic and comparisons. What each of these operations does is up to the programmer and should be carefully documented. For more methods and details, see <https://docs.python.org/3/reference/datamodel.html#special-method-names>.

Problem 3. Endow the `Backpack` class with two additional magic methods:

1. The `__eq__()` magic method is used to determine if two objects are equal, and is invoked by the `==` operator. Implement the `__eq__()` magic method for the `Backpack` class so that two `Backpack` objects are equal if and only if they have the same name, color, and number of contents.
2. The `__str__()` magic method returns the string representation of an object. This method is invoked by `str()` and used by `print()`. Implement the `__str__()` method in the `Backpack` class so that printing a `Backpack` object yields the following output (that is, construct and return the following string).

```
Owner:      <name>
Color:      <color>
Size:       <number of items in contents>
Max Size:   <max_size>
Contents:   [<item1>, <item2>, ...]
```

(Hint: Use the tab and newline characters `'\t'` and `'\n'` to align output nicely.)

Achtung!

Magic methods for comparison are **not** automatically related. For example, even though the `Backpack` class implements the magic methods for `<` and `==`, two `Backpack` objects cannot respond to the `<=` operator unless `__le__()` is explicitly defined. The exception to this rule is the `!=` operator: as long as `__eq__()` is defined, `A!=B` is `False` if and only if `A==B` is `True`.

Problem 4. Write a `ComplexNumber` class from scratch.

1. Complex numbers are denoted $a + bi$ where $a, b \in \mathbb{R}$ and $i = \sqrt{-1}$. Write the constructor so it accepts two numbers. Store the first as `self.real` and the second as `self.imag`.
2. The *complex conjugate* of $a + bi$ is defined as $\overline{a + bi} = a - bi$. Write a `conjugate()` method that returns the object's complex conjugate as a new `ComplexNumber` object.
3. Add the following magic methods:
 - (a) Implement `__str__()` so that $a + bi$ is printed out as $(a+bj)$ for $b \geq 0$ and $(a-bj)$ for $b < 0$.
 - (b) The *magnitude* of $a + bi$ is $|a + bi| = \sqrt{a^2 + b^2}$. The `__abs__()` magic method determines the output of the built-in `abs()` function (absolute value). Implement `__abs__()` so that it returns the magnitude of the complex number.
 - (c) Two `ComplexNumber` objects are equal if and only if they have the same real and imaginary parts. Implement `__eq__()` so that it compares the `ComplexNumber` object with another `ComplexNumber` object and returns a bool indicating whether they are equal.
 - (d) Implement `__add__()`, `__sub__()`, `__mul__()`, and `__truediv__()` appropriately. Each of these should return a new `ComplexNumber` object.

Write a function to test your class by comparing it to Python's built-in `complex` type.

```
def test_ComplexNumber(a, b):
    py_cnum, my_cnum = complex(a, b), ComplexNumber(a, b)

    # Validate the constructor.
    if my_cnum.real != a or my_cnum.imag != b:
        print("__init__() set self.real and self.imag incorrectly")

    # Validate conjugate() by checking the new number's imag attribute.
    if py_cnum.conjugate().imag != my_cnum.conjugate().imag:
        print("conjugate() failed for", py_cnum)

    # Validate __str__().
    if str(py_cnum) != str(my_cnum):
        print("__str__() failed for", py_cnum)
```

```
# ...
```

Additional Material

Static Attributes

Attributes that are accessed through `self` are called *instance* attributes because they are bound to a particular instance of the class. In contrast, a *static* attribute is one that is shared between all instances of the class. To make an attribute static, declare it inside of the `class` block but outside of any of the class's methods, and do not use `self`. Since the attribute is not tied to a specific instance of the class, it may be accessed or changed via the class name without even instantiating the class at all.

```
class Backpack:
    # ...
    brand = "Adidas"                # Backpack.brand is a static attribute.
```

```
>>> pack1, pack2 = Backpack("Bill", "blue"), Backpack("William", "white")
>>> print(pack1.brand, pack2.brand, Backpack.brand)
Adidas Adidas Adidas

# Change the brand name for the class to change it for all class instances.
>>> Backpack.brand = "Nike"
>>> print(pack1.brand, pack2.brand, Backpack.brand)
Nike Nike Nike
```

Static Methods

Individual class methods can also be static. A static method cannot be dependent on the attributes of individual instances of the class, so there can be no references to `self` inside the body of the method and `self` is **not** listed as an argument in the function definition. Thus static methods only have access to static attributes and other static methods. Include the tag `@staticmethod` above the function definition to designate a method as static.

```
class Backpack:
    # ...
    @staticmethod
    def origin():                    # Do not use 'self' as a parameter.
        print("Manufactured by " + Backpack.brand + ", inc.")
```

```
# Static methods can be called without instantiating the class.
>>> Backpack.origin()
Manufactured by Nike, inc.

# The method can also be accessed by individual class instances.
>>> pack = Backpack("Larry", "lime")
>>> pack.origin()
Manufactured by Nike, inc.
```

To practice these principles, consider adding a static attribute to the `Backpack` class to serve as a counter for a unique ID. In the constructor for the `Backpack` class, add an instance variable called `self.ID`. Set this ID based on the static ID variable, then increment the static ID so that the next `Backpack` object will have a different ID.

More Magic Methods

Consider how the following methods might be implemented for the `Backpack` class. These methods are particularly important for custom data structure classes.

Method	Operation	Trigger Function
<code>__bool__()</code>	Truth value	<code>bool()</code>
<code>__len__()</code>	Object length or size	<code>len()</code>
<code>__repr__()</code>	Object representation	<code>repr()</code>
<code>__getitem__()</code>	Indexing and slicing	<code>self[index]</code>
<code>__setitem__()</code>	Assignment via indexing	<code>self[index] = x</code>
<code>__iter__()</code>	Iteration over the object	<code>iter()</code>
<code>__reversed__()</code>	Reverse iteration over the object	<code>reversed()</code>
<code>__contains__()</code>	Membership testing	<code>in</code>

See <https://docs.python.org/3/reference/datamodel.html#special-method-names> for more details and documentation on all magic methods.

Hashing

A *hash value* is an integer that uniquely identifies an object. The built-in `hash()` function calculates an object's hash value by calling its `__hash__()` magic method.

In Python, the built-in `set` and `dict` structures use hash values to store and retrieve objects in memory quickly. If an object is *unhashable*, it cannot be put in a set or be used as a key in a dictionary. See <https://docs.python.org/3/glossary.html#term-hashable> for details.

If the `__hash__()` method is not defined, the default hash value is the object's memory address (accessible via the built-in function `id()`) divided by 16, rounded down to the nearest integer. However, two objects that compare as equal via the `__eq__()` magic method must have the same hash value. The following simple `__hash__()` method for the `Backpack` class conforms to this rule and returns an integer.

```
class Backpack:
    # ...
    def __hash__(self):
        return hash(self.name) ^ hash(self.color) ^ hash(len(self.contents))
```

The caret operator `^` is a bitwise XOR (exclusive or). The bitwise AND operator `&` and the bitwise OR operator `|` are also good choices to use.

See https://docs.python.org/3/reference/datamodel.html#object.__hash__ for more on hashing.

6

Introduction to Matplotlib

Lab Objective: *Matplotlib is the most commonly used data visualization library in Python. Being able to visualize data helps to determine patterns and communicate results and is a key component of applied and computational mathematics. In this lab we introduce techniques for visualizing data in 1, 2, and 3 dimensions. The plotting techniques presented here will be used in the remainder of the labs in the manual.*

Line Plots

Raw numerical data is rarely helpful unless it can be visualized. The quickest way to visualize a simple 1-dimensional array is with a *line plot*. The following code creates an array of outputs of the function $f(x) = x^2$, then visualizes the array using the `matplotlib` module¹ [?].

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt

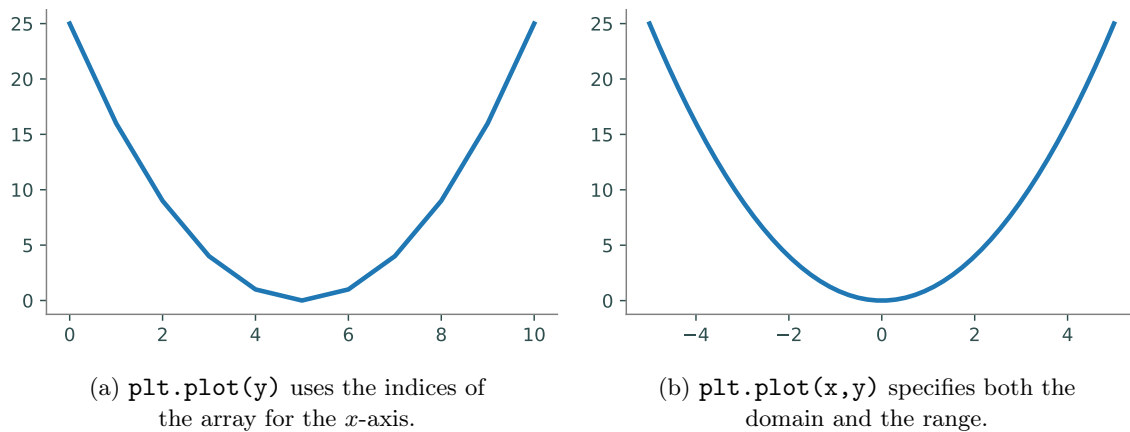
>>> y = np.arange(-5,6)**2
>>> y
array([25, 16,  9,  4,  1,  0,  1,  4,  9, 16, 25])

# Visualize the plot.
>>> plt.plot(y)                                # Draw the line plot.
[<matplotlib.lines.Line2D object at 0x1084762d0>]
>>> plt.show()                                # Reveal the resulting plot.
```

The result is shown in Figure 6.1a. Just as `np` is a standard alias for NumPy, `plt` is a standard alias for `matplotlib.pyplot` in the Python community.

The call `plt.plot(y)` creates a figure and draws straight lines connecting the entries of `y` relative to the *y*-axis. The *x*-axis is (by default) the index of the array, which in this case is the integers from 0 to 10. Calling `plt.show()` then displays the figure.

¹Like NumPy, Matplotlib is not part of the Python standard library, but it is included in most Python distributions. See <https://matplotlib.org/> for the complete Matplotlib documentation.

Figure 6.1: Plots of $f(x) = x^2$ over the interval $[-5, 5]$.

Problem 1. NumPy's `random` module has tools for sampling from probability distributions. For instance, `np.random.normal()` draws samples from the normal (Gaussian) distribution. The `size` parameter specifies the shape of the resulting array.

```
>>> np.random.normal(size=(2,3))    # Get a 2x3 array of samples.
array([[ 1.65896515, -0.43236783, -0.99390897],
       [-0.35753688, -0.76738306,  1.29683025]])
```

Write a function that accepts an integer n as input.

1. Use `np.random.normal()` to create an $n \times n$ array of values randomly sampled from the standard normal distribution.
2. Compute the mean of each row of the array.
(Hint: Use `np.mean()` and specify the `axis` keyword argument.)
3. Return the variance of these means.
(Hint: Use `np.var()` to calculate the variance).

Define another function that creates an array of the results of the first function with inputs $n = 100, 200, \dots, 1000$. Plot (and show) the resulting array.

Specifying a Domain

An obvious problem with Figure 6.1a is that the x -axis does not correspond correctly to the y -axis for the function $f(x) = x^2$ that is being drawn. To correct this, define an array `x` for the domain, then use it to calculate the image $y = f(x)$. The command `plt.plot(x,y)` plots `x` against `y` by drawing a line between the consecutive points `(x[i], y[i])`.

Another problem with Figure 6.1a is its poor resolution: the curve is visibly bumpy, especially near the bottom of the curve. NumPy's `linspace()` function makes it easy to get a higher-resolution domain. Recall that `np.arange()` returns an array of evenly-spaced values in a given interval, where

the **spacing** between the entries is specified. In contrast, `np.linspace()` creates an array of evenly-spaced values in a given interval where the **number of elements** is specified.

```
# Get 4 evenly-spaced values between 0 and 32 (including endpoints).
>>> np.linspace(0, 32, 4)
array([ 0.          , 10.66666667, 21.33333333, 32.          ])

# Get 50 evenly-spaced values from -5 to 5 (including endpoints).
>>> x = np.linspace(-5, 5, 50)
>>> y = x**2                                # Calculate the image of f(x) = x**2.
>>> plt.plot(x, y)
>>> plt.show()
```

The resulting plot is shown in Figure 6.1b. This time, the x -axis correctly matches up with the y -axis. The resolution is also much better because x and y have 50 entries each instead of only 10.

Subsequent calls to `plt.plot()` modify the same figure until `plt.show()` is executed, which displays the current figure and resets the system. This behavior can be altered by specifying separate figures or axes, which we will discuss shortly.

Note

Plotting can seem a little mystical because the actual plot doesn't appear until `plt.show()` is executed. Matplotlib's interactive mode allows the user to see the plot be constructed one piece at a time. Use `plt.ion()` to turn interactive mode on and `plt.ioff()` to turn it off. This is very useful for quick experimentation. Try executing the following commands in IPython:

```
In [1]: import numpy as np
In [2]: from matplotlib import pyplot as plt

# Turn interactive mode on and make some plots.
In [3]: plt.ion()
In [4]: x = np.linspace(1, 4, 100)
In [5]: plt.plot(x, np.log(x))
In [6]: plt.plot(x, np.exp(x))

# Clear the figure, then turn interactive mode off.
In [7]: plt.clf()
In [8]: plt.ioff()
```

Use interactive mode **only** with IPython. Using interactive mode in a non-interactive setting may freeze the window or cause other problems.

Problem 2. Write a function that plots the functions $\sin(x)$, $\cos(x)$, and $\arctan(x)$ on the domain $[-2\pi, 2\pi]$ (use `np.pi` for π). Make sure the domain is refined enough to produce a figure with good resolution.

Plot Customization

`plt.plot()` receives several keyword arguments for customizing the drawing. For example, the color and style of the line are specified by the following string arguments.

Key	Color	Key	Style
'b'	blue	'-'	solid line
'g'	green	'--'	dashed line
'r'	red	'-.'	dash-dot line
'c'	cyan	':'	dotted line
'k'	black	'o'	circle marker

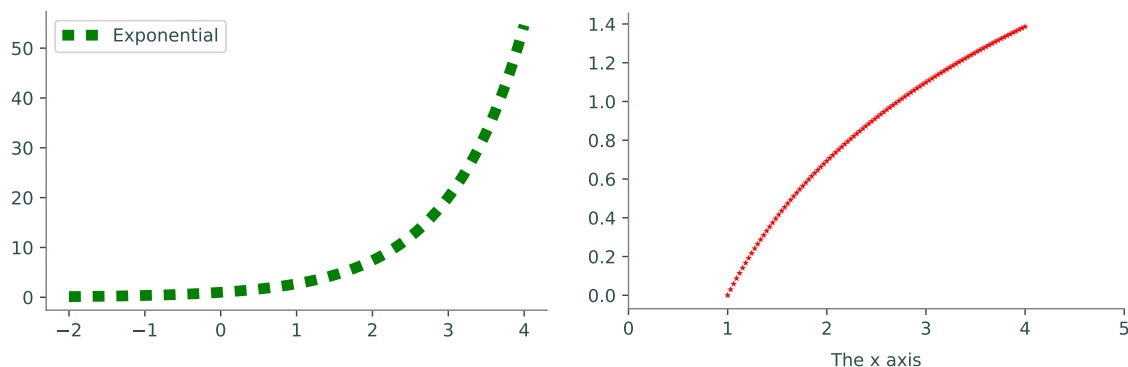
Specify one or both of these string codes as the third argument to `plt.plot()` to change from the default color and style. Other `plt` functions further customize a figure.

Function	Description
<code>legend()</code>	Place a legend in the plot
<code>title()</code>	Add a title to the plot
<code>xlim()</code> / <code>ylim()</code>	Set the limits of the <i>x</i> - or <i>y</i> -axis
<code>xlabel()</code> / <code>ylabel()</code>	Add a label to the <i>x</i> - or <i>y</i> -axis

```
>>> x1 = np.linspace(-2, 4, 100)
>>> plt.plot(x1, np.exp(x1), 'g:', linewidth=6, label="Exponential")
>>> plt.title("This is the title.", fontsize=18)
>>> plt.legend(loc="upper left")      # plt.legend() uses the 'label' argument of
>>> plt.show()                      # plt.plot() to create a legend.

>>> x2 = np.linspace(1, 4, 100)
>>> plt.plot(x2, np.log(x2), 'r*', markersize=4)
>>> plt.xlim(0, 5)                  # Set the visible limits of the x axis.
>>> plt.xlabel("The x axis")        # Give the x axis a label.
>>> plt.show()
```

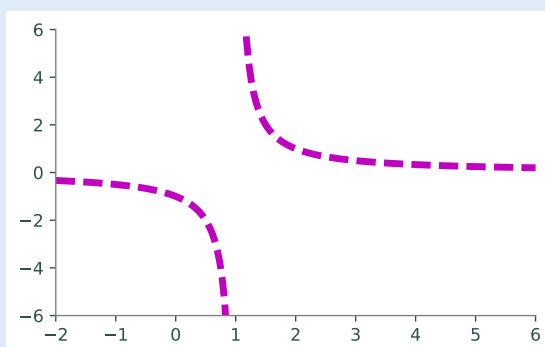
This is the title.



Problem 3. Write a function to plot the curve $f(x) = \frac{1}{x-1}$ on the domain $[-2, 6]$.

1. Although $f(x)$ has a discontinuity at $x = 1$, a single call to `plt.plot()` in the usual way will make the curve look continuous. Split up the domain into $[-2, 1)$ and $(1, 6]$. Plot the two sides of the curve separately so that the graph looks discontinuous at $x = 1$.
2. Plot both curves with a dashed magenta line. Set the keyword argument `linewidth` (or `lw`) of `plt.plot()` to 4 to make the line a little thicker than the default setting.
3. Use `plt.xlim()` and `plt.ylim()` to change the range of the x -axis to $[-2, 6]$ and the range of the y -axis to $[-6, 6]$.

The plot should resemble the figure below.



Figures, Axes, and Subplots

The window that `plt.show()` reveals is called a *figure*, stored in Python as a `plt.Figure` object. A space on a figure where a plot is drawn is called an *axes*, a `plt.Axes` object. A figure can have multiple axes, and a single program may create several figures. There are several ways to create or grab figures and axes with `plt` functions.

Function	Description
<code>axes()</code>	Add an axes to the current figure
<code>figure()</code>	Create a new figure or grab an existing figure
<code>gca()</code>	Get the current axes
<code>gcf()</code>	Get the current figure
<code>subplot()</code>	Add a single subplot to the current figure
<code>subplots()</code>	Create a figure and add several subplots to it

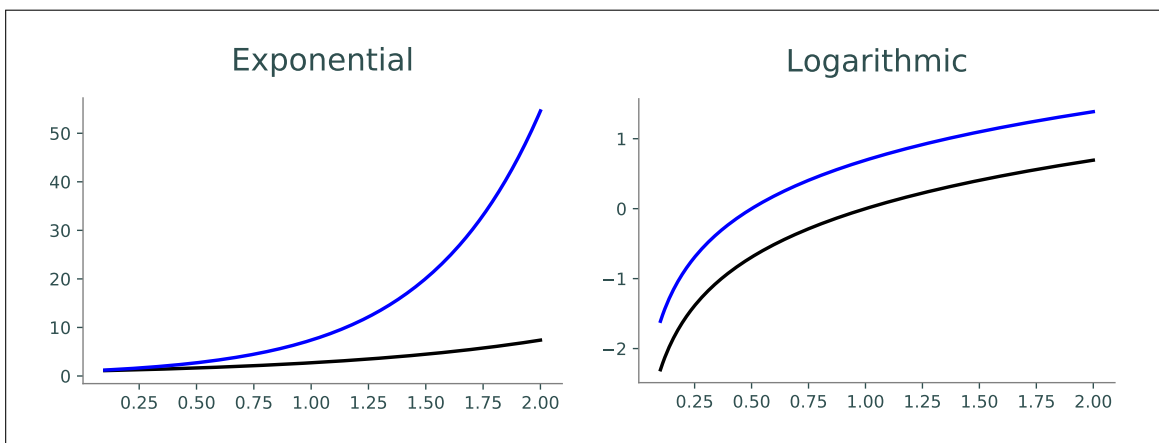
Usually when a figure has multiple axes, they are organized into non-overlapping *subplots*. The command `plt.subplot(nrows, ncols, plot_number)` creates an axes in a subplot grid where `nrows` is the number of rows of subplots in the figure, `ncols` is the number of columns, and `plot_number` specifies which subplot to modify. If the inputs for `plt.subplot()` are all integers, the commas between the entries can be omitted. For example, `plt.subplot(3,2,2)` can be shortened to `plt.subplot(322)`.



Figure 6.3: The layout of subplots with `plt.subplot(2,3,i)` (2 rows, 3 columns), where `i` is the index pictured above. The outer border is the figure that the axes belong to.

```
>>> x = np.linspace(.1, 2, 200)
# Create a subplot to cover the left half of the figure.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, np.exp(x), 'k', lw=2)
>>> ax1.plot(x, np.exp(2*x), 'b', lw=2)
>>> plt.title("Exponential", fontsize=18)

# Create another subplot to cover the right half of the figure.
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, np.log(x), 'k', lw=2)
>>> ax2.plot(x, np.log(2*x), 'b', lw=2)
>>> ax2.set_title("Logarithmic", fontsize=18)
>>> plt.show()
```



Note

Plotting functions such as `plt.plot()` are shortcuts for accessing the current axes on the current figure and calling a method on that `Axes` object. Calling `plt.subplot()` changes the current axis, and calling `plt.figure()` changes the current figure. Use `plt.gca()` to get the current axes and `plt.gcf()` to get the current figure. Compare the following equivalent strategies for producing a figure with two subplots.

```
>>> x = np.linspace(-5, 5, 100)

# 1. Use plt.subplot() to switch the current.
>>> plt.subplot(121)
>>> plt.plot(x, 2*x)
>>> plt.subplot(122)
>>> plt.plot(x, x**2)

# 2. Use plt.subplot() to explicitly grab the two subplot axes.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, 2*x)
>>> ax2 = plt.subplot(122)
>>> ax2.plot(x, x**2)

# 3. Use plt.subplots() to get the figure and all subplots simultaneously.
>>> fig, axes = plt.subplots(1, 2)
>>> axes[0].plot(x, 2*x)
>>> axes[1].plot(x, x**2)
```

Problem 4. Write a function that plots the functions $\sin(x)$, $\sin(2x)$, $2\sin(x)$, and $2\sin(2x)$ on the domain $[0, 2\pi]$, each in a separate subplot of a single figure.

1. Arrange the plots in a 2×2 grid of subplots.
2. Set the limits of each subplot to $[0, 2\pi] \times [-2, 2]$.
(Hint: Consider using `plt.axis([xmin, xmax, ymin, ymax])` instead of `plt.xlim()` and `plt.ylim()` to set all boundaries simultaneously.)
3. Use `plt.title()` or `ax.set_title()` to give each subplot an appropriate title.
4. Use `plt.suptitle()` or `fig.suptitle()` to give the overall figure a title.
5. Use the following colors and line styles.

$\sin(x)$: green solid line. $\sin(2x)$: red dashed line.

$2\sin(x)$: blue dashed line. $2\sin(2x)$: magenta dotted line.

Achtung!

Be careful not to mix up the following functions.

1. `plt.axes()` creates a new place to draw on the figure, while `plt.axis()` (or `ax.axis()`) sets properties of the x - and y -axis in the current axes, such as the x and y limits.
2. `plt.subplot()` (singular) returns a single subplot belonging to the current figure, while `plt.subplots()` (plural) creates a new figure and adds a collection of subplots to it.

Other Kinds of Plots

Line plots are not always the most illuminating choice of graph to describe a set of data. Matplotlib provides several other easy ways to visualize data.

- A *scatter plot* plots two 1-dimensional arrays against each other without drawing lines between the points. Scatter plots are particularly useful for data that is not correlated or ordered.

To create a scatter plot, use `plt.plot()` and specify a point marker (such as `'o'` or `'*'`) for the line style, or use `plt.scatter()` (or `ax.scatter()`). Beware that `plt.scatter()` has slightly different arguments and syntax than `plt.plot()`.

- A *histogram* groups entries of a 1-dimensional data set into a given number of intervals, called *bins*. Each bin has a bar whose height indicates the number of values that fall in the range of the bin. Histograms are best for displaying distributions, relating data values to frequency.

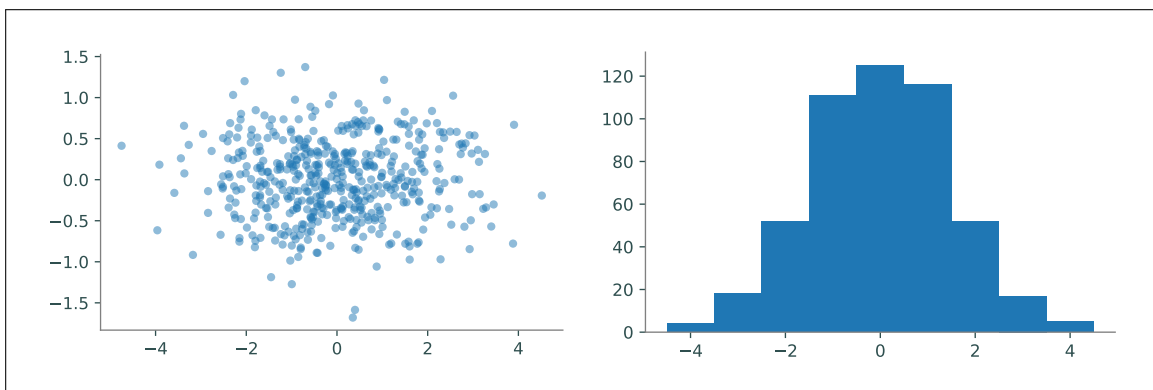
To create a histogram, use `plt.hist()` (or `ax.hist()`). Use the argument `bins` to specify the edges of the bins or to choose a number of bins. The `range` argument specifies the outer limits of the first and last bins.

```
# Get 500 random samples from two normal distributions.
>>> x = np.random.normal(scale=1.5, size=500)
>>> y = np.random.normal(scale=0.5, size=500)

# Draw a scatter plot of x against y, using transparent circle markers.
>>> ax1 = plt.subplot(121)
>>> ax1.plot(x, y, 'o', markersize=5, alpha=.5)

# Draw a histogram to display the distribution of the data in x.
>>> ax2 = plt.subplot(122)
>>> ax2.hist(x, bins=np.arange(-4.5, 5.5))      # Or, equivalently,
#   ax2.hist(x, bins=9, range=[-4.5, 4.5])

>>> plt.show()
```



Problem 5. The Fatality Analysis Reporting System (FARS) is a nationwide census that provides yearly data regarding fatal injuries suffered in motor vehicle traffic crashes.^a The array contained in `FARS.npy` is a small subset of the FARS database from 2010–2014. Each of the 148,206 rows in the array represents a different car crash; the columns represent the hour (in military time, as an integer), the longitude, and the latitude, in that order.

Write a function to visualize the data in `FARS.npy`. Use `np.load()` to load the data, then create a single figure with two subplots:

1. A scatter plot of longitudes against latitudes. Because of the large number of data points, use black pixel markers (use `"k,"` as the third argument to `plt.plot()`). Label both axes using `plt.xlabel()` and `plt.ylabel()` (or `ax.set_xlabel()` and `ax.set_ylabel()`). (Hint: Use `plt.axis("equal")` or `ax.set_aspect("equal")` so that the x - and y -axis are scaled the same way.
2. A histogram of the hours of the day, with one bin per hour. Set the limits of the x -axis appropriately. Label the x -axis. You should be able to clearly see which hours of the day experience more traffic.

^aSee <http://www.nhtsa.gov/FARS>.

Matplotlib also has tools for creating other kinds of plots for visualizing 1-dimensional data, including bar plots and box plots. See the Matplotlib Appendix for examples and syntax.

Visualizing 3-D Surfaces

Line plots, histograms, and scatter plots are good for visualizing 1- and 2-dimensional data, including the domain and range of a function $f : \mathbb{R} \rightarrow \mathbb{R}$. However, visualizing 3-dimensional data or a function $g : \mathbb{R}^2 \rightarrow \mathbb{R}$ (two inputs, one output) requires a different kind of plot. The process is similar to creating a line plot but requires slightly more setup: first construct an appropriate domain, then calculate the image of the function on that domain.

NumPy's `np.meshgrid()` function is the standard tool for creating a 2-dimensional domain in the Cartesian plane. Given two 1-dimensional coordinate arrays, `np.meshgrid()` creates two corresponding coordinate matrices. See Figure 6.6.

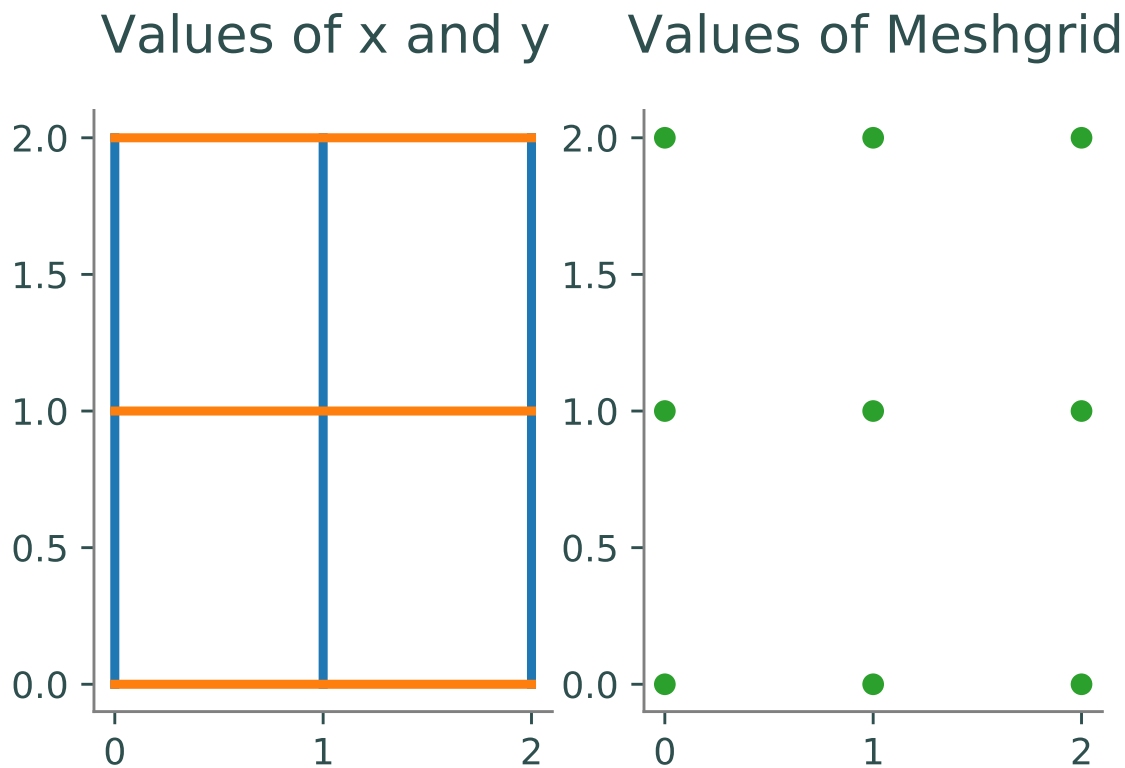


Figure 6.6: In the left plot, we have two arrays where x and y have the values $x = y = [0, 1, 2]$.

The command `np.meshgrid(x, y)` returns the arrays $X = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \\ 0 & 1 & 2 \end{bmatrix}$ and $Y = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$.

These give the x - and y -coordinates of the points in the grid formed by x and y as seen in the right plot and satisfy the possibly counterintuitive relation $(X[i, j], Y[i, j]) = (x[j], y[i])$.

```
>>> x, y = [0, 1, 2], [3, 4, 5]      # A rough domain over [0,2]x[3,5].
>>> X, Y = np.meshgrid(x, y)         # Combine the 1-D data into 2-D data.
>>> for xrow, yrow in zip(X, Y):
...     print(xrow, yrow, sep='\t')
...
[0 1 2]    [3 3 3]
[0 1 2]    [4 4 4]
[0 1 2]    [5 5 5]
```

With a 2-dimensional domain, $g(x, y)$ is usually visualized with two kinds of plots.

- A *heat map* assigns a color to each point in the domain, producing a 2-dimensional colored picture describing a 3-dimensional shape. Darker colors typically correspond to lower values while lighter colors typically correspond to higher values.

Use `plt.pcolormesh()` to create a heat map. You can add an optional argument for the shading type; this determines the layout and fill style of the heat map. This argument defaults to

`shading='auto'`, and will automatically choose a fill method suited to the data being graphed.

- A *contour map* draws several *level curves* of g on the 2-dimensional domain. A level curve corresponding to the constant c is the collection of points $\{(x, y) \mid c = g(x, y)\}$. Coloring the space between the level curves produces a discretized version of a heat map. Including more and more level curves makes a filled contour plot look more and more like the complete, blended heat map.

Use `plt.contour()` to create a contour plot and `plt.contourf()` to create a filled contour plot. Specify either the number of level curves to draw, or a list of constants corresponding to specific level curves.

These functions each receive the keyword argument `cmap` to specify a color scheme (some of the better schemes are "`viridis`", "`magma`", and "`coolwarm`"). For the list of all Matplotlib color schemes, see http://matplotlib.org/examples/color/colormaps_reference.html.

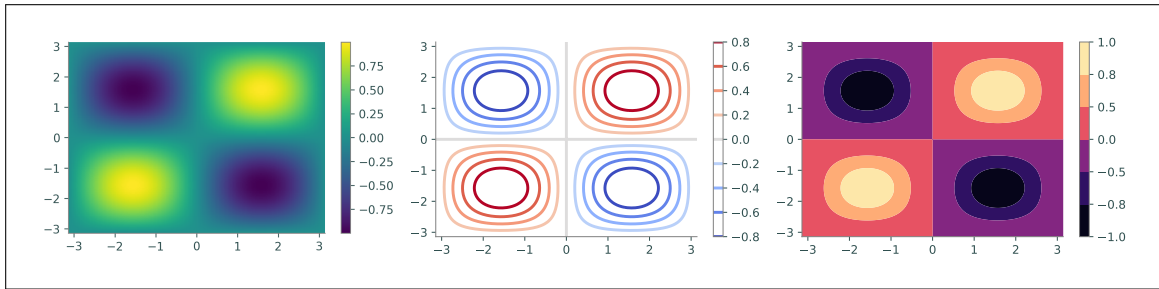
Finally, `plt.colorbar()` draws the color scale beside the plot to indicate how the colors relate to the values of the function.

```
# Create a 2-D domain with np.meshgrid().
>>> x = np.linspace(-np.pi, np.pi, 100)
>>> y = x.copy()
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)      # Calculate g(x,y) = sin(x)sin(y).

# Plot the heat map of f over the 2-D domain.
>>> plt.subplot(131)
>>> plt.pcolormesh(X, Y, Z, cmap="viridis", shading="auto")
>>> plt.colorbar()
>>> plt.xlim(-np.pi, np.pi)
>>> plt.ylim(-np.pi, np.pi)

# Plot a contour map of f with 10 level curves.
>>> plt.subplot(132)
>>> plt.contour(X, Y, Z, 10, cmap="coolwarm")
>>> plt.colorbar()

# Plot a filled contour map, specifying the level curves.
>>> plt.subplot(133)
>>> plt.contourf(X, Y, Z, [-1, -.8, -.5, 0, .5, .8, 1], cmap="magma")
>>> plt.colorbar()
>>> plt.show()
```



Problem 6. Write a function to plot $g(x, y) = \frac{\sin(x)\sin(y)}{xy}$ on the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.

1. Create 2 subplots: one with a heat map of g , and one with a contour map of g . Choose an appropriate number of level curves, or specify the curves yourself.
2. Set the limits of each subplot to $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$.
3. Choose a non-default color scheme.
4. Include a color scale bar for each subplot.

Additional Material

Further Reading and Tutorials

Plotting takes some getting used to. See the following materials for more examples.

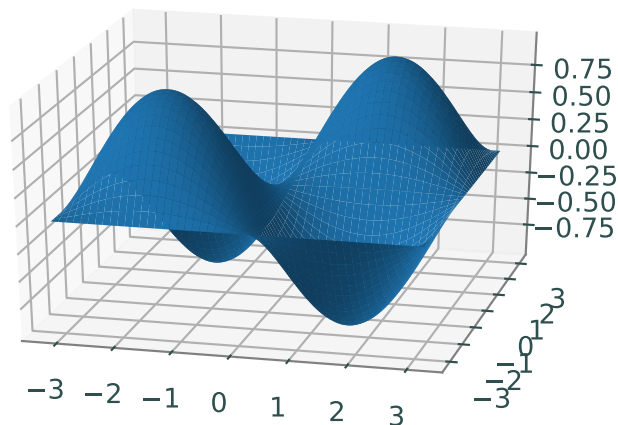
- <https://www.labri.fr/perso/nrougier/teaching/matplotlib/>.
- https://matplotlib.org/users/pyplot_tutorial.html.
- <http://scipy-lectures.org/intro/matplotlib/>.
- The Matplotlib Appendix in this manual.

3-D Plotting

Matplotlib can also be used to plot 3-dimensional surfaces. The following code produces the surface corresponding to $g(x, y) = \sin(x) \sin(y)$.

```
# Create the domain and calculate the range like usual.
>>> x = np.linspace(-np.pi, np.pi, 200)
>>> y = np.copy(x)
>>> X, Y = np.meshgrid(x, y)
>>> Z = np.sin(X) * np.sin(Y)

# Draw the corresponding 3-D plot using some extra tools.
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1, projection='3d')
>>> ax.plot_surface(X, Y, Z)
>>> plt.show()
```



Animations

Lines and other graphs can be altered dynamically to produce animations. Follow these steps to create a Matplotlib animation:

1. Calculate all data that is needed for the animation.
2. Define a figure explicitly with `plt.figure()` and set its window boundaries.
3. Draw empty objects that can be altered dynamically.
4. Define a function to update the drawing objects.
5. Use `matplotlib.animation.FuncAnimation()`.

The submodule `matplotlib.animation` contains the tools for putting together and managing animations. The function `matplotlib.animation.FuncAnimation()` accepts the figure to animate, the function that updates the figure, the number of frames to show before repeating, and how fast to run the animation (lower numbers mean faster animations).

```
from matplotlib.animation import FuncAnimation

def sine_animation():
    # Calculate the data to be animated.
    x = np.linspace(0, 2*np.pi, 200)[:~1]
    y = np.sin(x)

    # Create a figure and set the window boundaries of the axes.
    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    # Draw an empty line. The comma after 'drawing' is crucial.
    drawing, = plt.plot([], [])

    # Define a function that updates the line data.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        return drawing, # Note the comma!

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()
```

Try using the following function in place of `update()`. Can you explain why this animation is different from the original?

```
def wave(index):
    drawing.set_data(x, np.roll(y, index))
    return drawing,
```

To animate multiple objects at once, define the objects separately and make sure the update function returns both objects.

```

def sine_cosine_animation():
    x = np.linspace(0, 2*np.pi, 200)[: -1]
    y1, y2 = np.sin(x), np.cos(x)

    fig = plt.figure()
    plt.xlim(0, 2*np.pi)
    plt.ylim(-1.2, 1.2)

    sin_drawing, = plt.plot([], [])
    cos_drawing, = plt.plot([], [])

    def update(index):
        sin_drawing.set_data(x[:index], y1[:index])
        cos_drawing.set_data(x[:index], y2[:index])
        return sin_drawing, cos_drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10)
    plt.show()

```

Animations can also be 3-dimensional. The only major difference is an extra operation to set the 3-dimensional component of the drawn object. The code below animates the space curve parametrized by the following equations:

$$x(\theta) = \cos(\theta) \cos(6\theta), \quad y(\theta) = \sin(\theta) \cos(6\theta), \quad z(\theta) = \frac{\theta}{10}$$

```

def rose_animation_3D():
    theta = np.linspace(0, 2*np.pi, 200)
    x = np.cos(theta) * np.cos(6*theta)
    y = np.sin(theta) * np.cos(6*theta)
    z = theta / 10

    fig = plt.figure()
    ax = fig.gca(projection='3d')           # Make the figure 3-D.
    ax.set_xlim3d(-1.2, 1.2)              # Use ax instead of plt.
    ax.set_ylim3d(-1.2, 1.2)
    ax.set_aspect("equal")

    drawing, = ax.plot([], [], [])        # Provide 3 empty lists.

    # Update the first 2 dimensions like usual, then update the 3-D component.
    def update(index):
        drawing.set_data(x[:index], y[:index])
        drawing.set_3d_properties(z[:index])
        return drawing,

    a = FuncAnimation(fig, update, frames=len(x), interval=10, repeat=False)
    plt.show()

```


7

Exceptions and File Input/Output

Lab Objective: *In Python, an exception is an error detected during execution. Exceptions are important for regulating program usage and for correctly reporting problems to the programmer and end user. An understanding of exceptions is essential to safely read data from and write data to external files. Being able to interact with external files is important for analyzing data and communicating results. In this lab we learn exception syntax and file interaction protocols.*

Exceptions

An *exception* formally indicates an error and terminates the program early. Some of the more common exception types are listed below, along with the kinds of problems they typically indicate.

Exception	Indication
<code>AttributeError</code>	An attribute reference or assignment failed.
<code>ImportError</code>	An <code>import</code> statement failed.
<code>IndexError</code>	A sequence subscript was out of range.
<code>NameError</code>	A local or global name was not found.
<code>TypeError</code>	An operation or function was applied to an object of inappropriate type.
<code>ValueError</code>	An operation or function received an argument that had the right type but an inappropriate value.
<code>ZeroDivisionError</code>	The second argument of a division or modulo operation was zero.

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

>>> [1, 2, 3].fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'
```

Raising Exceptions

Most exceptions are due to coding mistakes and typos. However, exceptions can also be used intentionally to indicate a problem to the user or programmer. To create an exception, use the keyword `raise`, followed by the name of the exception class. As soon as an exception is raised, the program stops running unless the exception is handled properly.

```
>>> if 7 is not 7.0:                # Raise an exception with an error message.
...     raise Exception("ints and floats are different!")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: ints and floats are different!

>>> for x in range(10):
...     if x > 5:                    # Raise a specific kind of exception.
...         raise ValueError("'x' should not exceed 5.")
...     print(x, end=' ')
...
0 1 2 3 4 5
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
```

Problem 1. Consider the following arithmetic “magic” trick.

1. Choose a 3-digit number where the first and last digits differ by 2 or more (say, 123).
2. Reverse this number by reading it backwards (321).
3. Calculate the positive difference of these numbers ($321 - 123 = 198$).
4. Add the reverse of the result to itself ($198 + 891 = 1089$).

The result of the last step will always be 1089, regardless of the original number chosen in step 1 (can you explain why?).

The following function prompts the user for input at each step of the magic trick, but does not check that the user’s inputs are correct.

```
def arithmagic():
    step_1 = input("Enter a 3-digit number where the first and last "
                  "digits differ by 2 or more: ")
    step_2 = input("Enter the reverse of the first number, obtained "
                  "by reading it backwards: ")
    step_3 = input("Enter the positive difference of these numbers: ")
    step_4 = input("Enter the reverse of the previous result: ")
    print(str(step_3), "+", str(step_4), "= 1089 (ta-da!)")
```

Modify `arithmagic()` so that it verifies the user's input at each step. Raise a `ValueError` with an informative error message if any of the following occur:

- The first number (`step_1`) is not a 3-digit number.
- The first number's first and last digits differ by less than 2.
- The second number (`step_2`) is not the reverse of the first number.
- The third number (`step_3`) is not the positive difference of the first two numbers.
- The fourth number (`step_4`) is not the reverse of the third number.

(Hint: `input()` always returns a string, so each variable is a string initially. Use `int()` to cast the variables as integers when necessary. The built-in function `abs()` may also be useful.)

Handling Exceptions

To prevent an exception from halting the program, it must be handled by placing the problematic lines of code in a `try` block. An `except` block then follows with instructions for what to do in the event of an exception.

```
# The 'try' block should hold any lines of code that might raise an exception.
>>> try:
...     print("Entering try block...")
...     raise Exception("for no reason")
...     print("No problem!")           # This line gets skipped.
... # The 'except' block is executed just after the exception is raised.
... except Exception as e:
...     print("There was a problem:", e)
...
Entering try block...
There was a problem: for no reason
>>> # The program then continues on.
```

In this example, the name `e` represents the exception within the `except` block. Printing `e` displays its error message. If desired, `e` can be raised again with `raise e` or just `raise`.

The try-except control flow can be expanded with two other blocks, forming a code structure similar to a sequence of `if-elif-else` blocks.

1. The `try` block is executed until an exception is raised (if at all).
2. An `except` statement specifying the same kind of exception that was raised in the try block “catches” the exception, and the block is then executed. There may be multiple except blocks following a single try block (similar to having several `elif` statements following a single `if` statement), and a single except statement may specify multiple kinds of exceptions to catch.
3. The `else` block is executed if an exception was **not** raised in the try block.
4. The `finally` block is always executed if it is included.

```

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
...     raise ValueError("The house is on fire!")
...     # Check for multiple kinds of exceptions using parentheses.
... except (ValueError, TypeError) as e:
...     print("caught an exception.")
...     house_on_fire = True
... else:
...     # Skipped due to the exception.
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...caught an exception.
The house is on fire: True

>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
... except ValueError as e:
...     # Skipped because there was no exception.
...     print("caught a ValueError.")
...     house_on_fire = True
... except TypeError as e:
...     # Also skipped.
...     print("caught a TypeError.")
...     house_on_fire = True
... else:
...     print("no exceptions raised.")
... finally:
...     print("The house is on fire:", house_on_fire)
...
Entering try block...no exceptions raised.
The house is on fire: False

```

The code in the `finally` block is always executed, even if a `return` statement or an uncaught exception occurs in any block following the `try` statement.

```

>>> def implode():
...     try:
...         # Try to return immediately...
...         return
...     finally:
...         # ...but 'finally' goes before 'return'.
...         print("Goodbye, world!")
...
>>> implode()
Goodbye, world!

```

See <https://docs.python.org/3/tutorial/errors.html> for more examples.

Achtung!

An `except` statement with no specified exception type catches **any** exception raised in the corresponding `try` block. This approach can mistakenly mask unexpected errors. Always be specific about the kinds of exceptions you expect to encounter.

```
>>> def divider(x, y):
...     try:
...         return x / yy          # The misspelled yy raises a NameError.
...     except:                   # Catch ANY exception.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
y must not equal zero!

>>> def divider(x, y):
...     try:
...         return x / yy
...     except ZeroDivisionError: # Specify an exception type.
...         print("y must not equal zero!")
...
>>> divider(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in divider
NameError: name 'yy' is not defined      # Now the mistake is obvious.
```

Problem 2. A *random walk* is a path created by a sequence of random steps. The following function simulates a random walk by repeatedly adding or subtracting 1 to a running total.

```
from random import choice

def random_walk(max_iters=1e12):
    walk = 0
    directions = [1, -1]
    for i in range(int(max_iters)):
        walk += choice(directions)
    return walk
```

A `KeyboardInterrupt` is a special exception that can be triggered at any time by entering `ctrl+c` (on most systems) in the keyboard. Modify `random_walk()` so that if the user raises a `KeyboardInterrupt` by pressing `ctrl+c` while the program is running, the function catches the exception and prints “Process interrupted at iteration *i*”. If no `KeyboardInterrupt` is raised, print “Process completed”. In both cases, return `walk` as before.

The With Statement

An `IOError` indicates that some input or output operation has failed. A simple `try-finally` control flow can ensure that a file stream is closed safely.

The `with` statement provides an alternative method for safely opening and closing files. Use `with open(<filename>, <mode>) as <alias>:` to create an indented block in which the file is open and available under the specified alias. At the end of the block, the file is automatically and safely closed, even in the event of an exception. This is the preferred file-reading method when a file only needs to be accessed briefly.

```
>>> myfile = open("hello_world.txt", 'r')    # Open a file for reading.
>>> try:
...     contents = myfile.readlines()        # Read in the content by line.
... finally:
...     myfile.close()                      # Explicitly close the file.

# Equivalently, use a 'with' statement to take care of errors.
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...                                     # The file is closed automatically.
```

In both cases, if the file `hello_world.txt` does not exist in the current directory, `open()` raises a `FileNotFoundError`. However, errors in the `try` or `with` blocks do not prevent the file from being safely closed.

Reading and Writing

Open file objects have an implicit *cursor* that determines the location in the file to read from or write to. After the entire file has been read once, either the file must be closed and reopened, or the cursor must be reset to the beginning of the file with `seek(0)` before it can be read again.

Some of the more important file object attributes and methods are listed below.

Attribute	Description
<code>closed</code>	<code>True</code> if the object is closed.
<code>mode</code>	The access mode used to open the file object.
<code>name</code>	The name of the file.
Method	Description
<code>close()</code>	Close the connection to the file.
<code>read()</code>	Read a given number of bytes; with no input, read the entire file.
<code>readline()</code>	Read a line of the file, including the newline character at the end.
<code>readlines()</code>	Call <code>readline()</code> repeatedly and return a list of the resulting lines.
<code>seek()</code>	Move the cursor to a new position.
<code>tell()</code>	Report the current position of the cursor.
<code>write()</code>	Write a single string to the file (spaces are not added).
<code>writelines()</code>	Write a list of strings to the file (newline characters are not added).

Only strings can be written to files; to write a non-string type, first cast it as a string with `str()`. Be mindful of spaces and newlines to separate the data.

```
>>> with open("out.txt", 'w') as outfile:    # Open 'out.txt' for writing.
...     for i in range(10):
...         outfile.write(str(i**2)+' ')    # Write some strings (and spaces).
...
>>> outfile.closed                          # The file is closed automatically.
True
```

Problem 3. Define a class called `ContentFilter`. Implement the constructor so that it accepts the name of a file to be read.

1. If the file name is invalid in any way, prompt the user for another filename using `input()`. Continue prompting the user until they provide a valid filename.

```
>>> cf1 = ContentFilter("hello_world.txt")    # File exists.
>>> cf2 = ContentFilter("not-a-file.txt")     # File doesn't exist.
Please enter a valid file name: still-not-a-file.txt
Please enter a valid file name: hello_world.txt
>>> cf3 = ContentFilter([1, 2, 3])           # Not even a string.
Please enter a valid file name: hello_world.txt
```

(Hint: `with open()` might raise a `FileNotFoundError`, a `TypeError`, or an `OSError`.)

2. Read the file and store its name and contents as attributes (store the contents as a single string). Make sure the file is securely closed.

String Formatting

The `str` class has several useful methods for parsing and formatting strings. They are particularly useful for processing data from a source file and for preparing data to be written to an external file.

Method	Returns
<code>count()</code>	The number of times a given substring occurs within the string.
<code>find()</code>	The lowest index where a given substring is found.
<code>isalpha()</code>	<code>True</code> if all characters in the string are alphabetic (a, b, c, ...).
<code>isdigit()</code>	<code>True</code> if all characters in the string are digits (0, 1, 2, ...).
<code>isspace()</code>	<code>True</code> if all characters in the string are whitespace (" ", '\t', '\n').
<code>join()</code>	The concatenation of the strings in a given iterable with a specified separator between entries.
<code>lower()</code>	A copy of the string converted to lowercase.
<code>upper()</code>	A copy of the string converted to uppercase.
<code>replace()</code>	A copy of the string with occurrences of a given substring replaced by a different specified substring.
<code>split()</code>	A list of segments of the string, using a given character or string as a delimiter.
<code>strip()</code>	A copy of the string with leading and trailing whitespace removed.

The `join()` method translates a list of strings into a single string by concatenating the entries of the list and placing the principal string between the entries. Conversely, `split()` translates the principal string into a list of substrings, with the separation determined by a single input.

```
# str.join() puts the string between the entries of a list.
>>> words = ["state", "of", "the", "art"]
>>> "-".join(words)
'state-of-the-art'

# str.split() creates a list out of a string, given a delimiter.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split('\n')
['One fish', 'Two fish', 'Red fish', 'Blue fish', '']

# If no delimiter is provided, the string is split by whitespace characters.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split()
['One', 'fish', 'Two', 'fish', 'Red', 'fish', 'Blue', 'fish']
```

Can you tell the difference between the following routines?

```
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.readlines()
...
>>> with open("hello_world.txt", 'r') as myfile:
...     contents = myfile.read().split('\n')
```

Variables within Strings

In addition to formatting strings using the methods above there may also be times when you wish to insert generalized variable values into the middle of strings. There are two ways to do this: using `.format()` or using f-strings. Of the two, f-strings are generally more readable, concise, faster, and less prone to error. As a result, an explanation of how to use `.format()` has been banished to the "Additional Materials" section, which we know is rarely read, and a short explanation of f-strings will be included here.

We define an f-string with an `'f'` preceding the string declaration (i.e `f'your words here'`, or `f"your words here"`). Within the f-string, curly brackets can be used along with variable names to insert the variable value into the string. Consider the following code which puts an argument defaulting to `'bread and butter pickles'` into the sentence: `"I think that (blank) are an abomination."` Note that f-strings can also be used with multiple variables at once by using multiple sets of curly brackets.

```
# defining function
>>> def abomination(userOpinion='Bread and Butter Pickles'):
...     # notice the use of the f-string
...     return f'I think that {userOpinion} are an abomination'

# calling the function
>>> abomination()
'I think that Bread and Butter Pickles are an abomination.'
```

Problem 4. Add the following methods to the `ContentFilter` class for writing the contents of the original file to new files. Each method should accept the name of a file to write to and a keyword argument `mode` that specifies the file access mode, defaulting to `'w'`. If `mode` is not `'w'`, `'x'`, or `'a'`, raise a `ValueError` with an informative message.

1. `uniform()`: write the data to the outfile with uniform case. Include an additional keyword argument `case` that defaults to `"upper"`.
If `case="upper"`, write the data in upper case. If `case="lower"`, write the data in lower case. If `case` is not one of these two values, raise a `ValueError`.
2. `reverse()`: write the data to the outfile in reverse order. Include an additional keyword argument `unit` that defaults to `"line"`.
If `unit="word"`, reverse the ordering of the words in each line, but write the lines in the same order as the original file. If `unit="line"`, reverse the ordering of the lines, but do not change the ordering of the words on each individual line. If `unit` is not one of these two values, raise a `ValueError`.
3. `transpose()`: write a “transposed” version of the data to the outfile. That is, write the first word of each line of the data to the first line of the new file, the second word of each line of the data to the second line of the new file, and so on. Viewed as a matrix of words, the rows of the input file then become the columns of the output file, and vice versa. You may assume that there are an equal number of words on each line of the input file.
4. `__str__()`: Also implement the `__str__()` magic method so that printing a `ContentFilter` object yields the following output. You may want to calculate these statistics in the constructor. (Note: Using f-strings will also make this implementation much simpler).

```
Source file:           <filename>
Total characters:     <The total number of characters in file>
Alphabetic characters: <The number of letters>
Numerical characters: <The number of digits>
Whitespace characters: <The number of spaces, tabs, and newlines>
Number of lines:      <The number of lines>
```

(Hint: list comprehensions are **very** useful for some of these functions. For example, what does `[line[:-1] for line in lines]` do? What about `sum([s.isspace() for s in data])`?)

Compare your class to the following example.

```
# cf_example1.txt
A b C
d E f
```

```
>>> cf = ContentFilter("cf_example1.txt")
>>> cf.uniform("uniform.txt", mode='w', case="upper")
>>> cf.uniform("uniform.txt", mode='a', case="lower")
>>> cf.reverse("reverse.txt", mode='w', unit="word")
>>> cf.reverse("reverse.txt", mode='a', unit="line")
>>> cf.transpose("transpose.txt", mode='w')
```

```
# uniform.txt
A B C
D E F
a b c
d e f
```

```
# reverse.txt
C b A
f E d
d E f
A b C
```

```
# transpose.txt
A d
b E
C f
```

Additional Material

Custom Exception Classes

Custom exceptions can be defined by writing a class that inherits from some existing exception class. The generic `Exception` class is typically the parent class of choice.

```
>>> class TooHardError(Exception):
...     pass
...
>>> raise TooHardError("This lab is impossible!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.TooHardError: This lab is impossible!
```

This may seem like a trivial extension of the `Exception` class, but it is useful to do because the interpreter never automatically raises a `TooHardError`. Any `TooHardError` must have originated from a hand-written `raise` command, making it easier to identify the exact source of the problem.

Chaining Exceptions

Sometimes, especially in large programs, it is useful to raise one kind of exception just after catching another. The two exceptions can be linked together using the `from` statement. This syntax makes it possible to see where the error originated from and to “pass it up” to another part of the program.

```
>>> try:
...     raise TooHardError("This lab is impossible!")
... except TooHardError as e:
...     raise NotImplementedError("Lab is incomplete") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
__main__.TooHardError: This lab is impossible!

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NotImplementedError: Lab is incomplete
```

More String Formatting Tools

Concatenating string values with non-string values can be cumbersome and tedious. The `str` class’s `format()` method makes it easier to insert non-string values into the middle of a string. Write the desired output in its entirety, replacing non-string values with curly braces `{}`. Then use the `format()` method, entering each replaced value in order.

```
# Join the data using string concatenation.
>>> day, month, year = 10, "June", 2017
```



```
>>> print("Is today", day, str(month) + ',', str(year) + "?")
Is today 10 June, 2017?

# Join the data using str.format().
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 10 June, 2017?

# Join the data easier using a f-string.
>>> print(f"Is today {day} {month}, {year}?")
Is today 10 June, 2017?
```

This method is extremely flexible and provides many convenient ways to format string output nicely. Consider the following code for printing out a simple progress bar from within a loop.

```
>>> iters = int(1e7)
>>> chunk = iters // 20
>>> for i in range(iters):
...     print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...                                           end='', flush=True)
...
...
```

Here the string `"\r[{:<20}]"` used in conjunction with the `format()` method tells the cursor to go back to the beginning of the line, print an opening bracket, then print the first argument of `format()` left-aligned with at least 20 total spaces before printing the closing bracket. The `end` parameter can change the default newline added to the output to another ending. Setting to an empty string, `end=''`, the output ends without any whitespace. The `flush` parameter does not need to be changed if the `end` parameter is not changed and it defaults to `False`. As shown in the example above, setting the parameter equal to `True` forces the output to be printed on the terminal before it is complete. If it is `False` with the `end=''` the `print()` function will first build the output with the set ending before printing each iteration rather than printing incrementally.

Printing at each iteration dramatically slows down the progression through the loop. How does the following code solve that problem?

```
>>> for i in range(iters):
...     if not i % chunk:
...         print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i),
...                                           end='', flush=True)
...
...
```

See <https://docs.python.org/3/library/string.html#format-string-syntax> for more examples and specific syntax for using `str.format()`. For a more robust progress bar printer, research the `tqdm` module.

Standard Library Modules for I/O

The standard library has other tools for input and output operations. For details on each module, see <https://docs.python.org/3/library>.

Module	Description
<code>csv</code>	CSV (comma separated value) file writing and parsing.
<code>io</code>	Support for file objects and <code>open()</code> .
<code>os</code>	Communication with the operating system.
<code>os.path</code>	Common path operations such as checking for file existence.
<code>pickle</code>	Create portable serialized representations of Python objects.

8

Unit Testing

Lab Objective: *Finding and fixing programming errors can be difficult and time consuming, especially in large or complex programs. Unit testing is a formal strategy for finding and eliminating errors quickly as a program is constructed and for ensuring that the program still works whenever it is modified. A single unit test checks a small piece code (usually a function or class method) for correctness, independent of the rest of the program. A well-written collection of unit tests can ensure that every unit of code functions as intended, thereby certifying that the program is correct. In this lab we learn to write unit tests in Python and practice test-driven development. Applying these principles will greatly speed up the coding process and improve your code quality.*

Unit Tests

A *unit test* verifies a piece of code by running a series of test cases and comparing actual outputs with expected outputs. Each test case is usually checked with an `assert` statement, a shortcut for raising an `AssertionError` with an optional error message if a boolean statement is false.

```
# Store the result of a boolean expression in a variable.
>>> result = str(5)=='5'

# Check the result, raising an error if it is false.
>>> if result is False:
...     raise AssertionError("incorrect result")

# Do the same check in one line with an assert statement.
>>> assert result, "incorrect result"

# Asserting a false statement raises an AssertionError.
>>> assert 5=='5', "5 is not a string"
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
AssertionError: 5 is not a string
```

Now suppose we wanted to test a simple `add()` function, located in the file `specs.py`.

```
# specs.py

def add(a, b):
    """Add two numbers."""
    return a + b
```

In a corresponding file called `test_specs.py`, which should contain all of the unit tests for the code in `specs.py`, we write a unit test called `test_add()` to verify the `add()` function.

```
# test_specs.py
import specs

def test_add():
    assert specs.add(1, 3) == 4, "failed on positive integers"
    assert specs.add(-5, -7) == -12, "failed on negative integers"
    assert specs.add(-6, 14) == 8
```

In this case, running `test_add()` raises no errors since all three test cases pass. Unit test functions don't need to return anything, but they should raise an exception if a test case fails.

Note

This style of external testing—checking that certain inputs result in certain outputs—is called black box testing. The actual structure of the code is not considered, but what it produces is thoroughly examined. In fact, the author of a black box test doesn't even need to be the person who eventually writes the program: having one person write tests and another write the code helps detect problems that one developer or the other may not have caught individually.

PyTest

Python's `pytest` module¹ provides tools for building tests, running tests, and providing detailed information about the results. To begin, run `pytest` in the current directory. Without any test files, the output should be similar to the following.

```
$ pytest
===== test session starts =====
platform darwin -- Python 3.6.0, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/Student, inifile:
collected 0 items

===== no tests ran in 0.02 seconds =====
```

Given some test files, say `test_calendar.py` and `test_google.py`, the output of `pytest` identifies failed tests and provides details on why they failed.

¹Pytest is not part of the standard library, but it is included in Anaconda's Python distribution. Install `pytest` with `[basicstyle=]conda install pytest` if needed. The standard library's `[basicstyle=]unittest` module also provides a testing framework, but is less popular and straightforward than `PyTest`.

```

$ pytest
===== test session starts =====
platform darwin -- Python 3.6.0, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/Student/example_tests, inifile:
collected 12 items

test_calendar.py .....
test_google.py .F..

===== FAILURES =====
----- test_subtract -----

    def test_subtract():
>         assert google.subtract(42, 17)==25, "subtract() failed for a > b > 0"
E         AssertionError: subtract() failed for a > b > 0
E         assert 35 == 25
E         +   where 35 = <function subtract at 0x102d4eb90>(42, 17)
E         +   where <function subtract at 0x102d4eb90> = google.subtract

test_google.py:11: AssertionError
===== 1 failed, 11 passed in 0.02 seconds =====

```

Each dot represents a passed test and each F represents a failed test. They show up in order, so in the example above, only the second of four tests in `test_google.py` failed.

Achtung!

PyTest will not find or run tests if they are not contained in files named `test_*.py` or `*_test.py`, where `*` represents any number of characters. In addition, the unit tests themselves must be named `test_*()` or `*_test()`. If you need to change this behavior, consult the documentation at <http://pytest.org/latest/example/pythoncollection.html>.

Problem 1. The following function contains a subtle but important error.

```

def smallest_factor(n):
    """Return the smallest prime factor of the positive integer n."""
    if n == 1: return 1
    for i in range(2, int(n**.5)):
        if n % i == 0: return i
    return n

```

Write a unit test for this function, including test cases that you suspect might uncover the error (what are the edge cases for this function?). Use `pytest` to run your unit test and discover a test case that fails, then use this information to correct the function.

Coverage

Successful unit tests include enough test cases to test the entire program. *Coverage* refers to the number of lines of code that are executed by at least one test case. One tool for measuring coverage is called `pytest-cov`, an extension of `pytest`. This tool must be installed separately, as it does not come bundled with Anaconda. To install, run the following code in a terminal.

```
$ conda install pytest-cov
```

Add the flag `--cov` to the `pytest` command to print out code coverage information. Running `pytest --cov` in the same directory as `specs.py` and `test_specs.py` yields the following output.

```
$ pytest --cov
===== test session starts =====
platform darwin -- Python 3.6.0, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/Student/Testing, inifile:
plugins: cov-2.3.1
collected 7 items

test_specs.py .....

----- coverage: platform darwin, python 3.6.6-final-0 -----
Name                Stmts  Miss  Cover
-----
specs.py              73     34    53%
test_specs.py         46      0   100%
-----
TOTAL                 119     34    71%

===== 7 passed in 0.03 seconds =====
```

Here, `Stmts` refers to the number of lines of code covered by a unit test, while `Miss` is the number of lines that are not currently covered. Notice that the file `test_specs.py` has 100% coverage while `specs.py` does not. Test files generally have 100% coverage, since `pytest` is designed to run these files in their entirety. However, `specs.py` does not have full coverage and requires additional unit tests. To find out which lines are not yet covered, `pytest-cov` has a useful feature called `cov-report` that creates an HTML file for visualizing the current line coverage.

```
$ pytest --cov-report html --cov
===== test session starts =====
# ...
----- coverage: platform darwin, python 3.6.6-final-0 -----
Coverage HTML written to dir htmlcov
```

Instead of printing coverage statistics, this command creates various files with coverage details in a new directory called `htmlcov/`. The file `htmlcov/specs_py.html`, which can be viewed in an internet browser, highlights in red the lines of `specs.py` that are not yet covered by any unit tests.

Note

Statement coverage is categorized as white box testing because it requires an understanding of the code's structure. While most black box tests can be written before a program is actually implemented, white box tests should be added to the collection of unit tests after the program is completed. By designing unit tests so that they cover every statement in a program, you may discover that some lines of code are unreachable, find that a conditional statement isn't functioning as intended, or uncover problems that accompany edge cases.

Problem 2. With `pytest-cov` installed, check your coverage of `smallest_factor()` from Problem 1. Write additional test cases if necessary to get complete coverage. Then, write a comprehensive unit test for the following (correctly written) function.

```
def month_length(month, leap_year=False):
    """Return the number of days in the given month."""
    if month in {"September", "April", "June", "November"}:
        return 30
    elif month in {"January", "March", "May", "July",
                  "August", "October", "December"}:
        return 31
    if month == "February":
        if not leap_year:
            return 28
        else:
            return 29
    else:
        return None
```

Testing Exceptions

Many programs are designed to raise exceptions in response to bad input or an unexpected error. A good unit test makes sure that the program raises the exceptions that it is expected to raise, but also that it doesn't raise any unexpected exceptions. The `raises()` method in `pytest` is a clean, formal way of asserting that a program raises a desired exception.

```
# specs.py

def divide(a, b):
    """Divide two numbers, raising an error if the second number is zero."""
    if b == 0:
        raise ZeroDivisionError("second input cannot be zero")
    return a / b
```

The corresponding unit tests checks that the function raises the `ZeroDivisionError` correctly.

```
# test_specs.py
import pytest

def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    pytest.raises(ZeroDivisionError, specs.divide, a=4, b=0)
```

If calling `divide(a=4, b=0)` results in a `ZeroDivisionError`, `pytest.raises()` catches the exception and the test case passes. On the other hand, if `divide(a=4, b=0)` does not raise a `ZeroDivisionError`, or if it raises a different kind of exception, the test fails.

To ensure that the `ZeroDivisionError` is coming from the written `raise` statement, combine `pytest.raises()` and the `with` statement to check the exception's error message.

```
def test_divide():
    assert specs.divide(4,2) == 2, "integer division"
    assert specs.divide(5,4) == 1.25, "float division"
    with pytest.raises(ZeroDivisionError) as excinfo:
        specs.divide(4, 0)
    assert excinfo.value.args[0] == "second input cannot be zero"
```

Here `excinfo` is an object containing information about the exception; the actual exception object is stored in `excinfo.value`, and hence `excinfo.value.args[0]` is the error message.

Problem 3. Write a comprehensive unit test for the following function. Make sure that each exception is raised properly by explicitly checking the exception message. Use `pytest-cov` and its `cov-report` tool to confirm that you have full coverage for this function.

```
def operate(a, b, oper):
    """Apply an arithmetic operation to a and b."""
    if type(oper) is not str:
        raise TypeError("oper must be a string")
    elif oper == '+':
        return a + b
    elif oper == '-':
        return a - b
    elif oper == '*':
        return a * b
    elif oper == '/':
        if b == 0:
            raise ZeroDivisionError("division by zero is undefined")
        return a / b
    raise ValueError("oper must be one of '+', '/', '-', or '*'")
```


Fixtures

Consider the following class for representing rational numbers as reduced fractions.

```
class Fraction(object):
    """Reduced fraction class with integer numerator and denominator."""
    def __init__(self, numerator, denominator):
        if denominator == 0:
            raise ZeroDivisionError("denominator cannot be zero")
        elif type(numerator) is not int or type(denominator) is not int:
            raise TypeError("numerator and denominator must be integers")

        def gcd(a,b):
            while b != 0:
                a, b = b, a % b
            return a
        common_factor = gcd(numerator, denominator)
        self.number = numerator // common_factor
        self.denom = denominator // common_factor

    def __str__(self):
        if self.denom != 1:
            return "{} / {}".format(self.number, self.denom)
        else:
            return str(self.number)

    def __float__(self):
        return self.number / self.denom

    def __eq__(self, other):
        if type(other) is Fraction:
            return self.number==other.number and self.denom==other.denom
        else:
            return float(self) == other

    def __add__(self, other):
        return Fraction(self.number*other.number + self.denom*other.denom,
                        self.denom*other.denom)

    def __sub__(self, other):
        return Fraction(self.number*other.number - self.denom*other.denom,
                        self.denom*other.denom)

    def __mul__(self, other):
        return Fraction(self.number*other.number, self.denom*other.denom)

    def __truediv__(self, other):
        if self.denom*other.number == 0:
            raise ZeroDivisionError("cannot divide by zero")
        return Fraction(self.number*other.denom, self.denom*other.number)
```

```
>>> from specs import Fraction
>>> print(Fraction(8, 12))           # 8/12 reduces to 2/3.
2 / 3
>>> Fraction(1, 5) == Fraction(3, 15) # 3/15 reduces to 1/5.
True
>>> print(Fraction(1, 3) * Fraction(1, 4))
1 / 12
```

To test this class, it would be nice to have some ready-made `Fraction` objects to use in each unit test. A *fixture*, a function marked with the `@pytest.fixture` decorator, sets up variables that can be used as mock data for multiple unit tests. The individual unit tests take the fixture function in as input and unpack the constructed tests. Below, we define a fixture that instantiates three `Fraction` objects. The unit tests for the `Fraction` class use these objects as test cases.

```
@pytest.fixture
def set_up_fractions():
    frac_1_3 = specs.Fraction(1, 3)
    frac_1_2 = specs.Fraction(1, 2)
    frac_n2_3 = specs.Fraction(-2, 3)
    return frac_1_3, frac_1_2, frac_n2_3

def test_fraction_init(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_3.numer == 1
    assert frac_1_2.denom == 2
    assert frac_n2_3.numer == -2
    frac = specs.Fraction(30, 42)           # 30/42 reduces to 5/7.
    assert frac.numer == 5
    assert frac.denom == 7

def test_fraction_str(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert str(frac_1_3) == "1 / 3"
    assert str(frac_1_2) == "1 / 2"
    assert str(frac_n2_3) == "-2 / 3"

def test_fraction_float(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert float(frac_1_3) == 1 / 3.
    assert float(frac_1_2) == .5
    assert float(frac_n2_3) == -2 / 3.

def test_fraction_eq(set_up_fractions):
    frac_1_3, frac_1_2, frac_n2_3 = set_up_fractions
    assert frac_1_2 == specs.Fraction(1, 2)
    assert frac_1_3 == specs.Fraction(2, 6)
    assert frac_n2_3 == specs.Fraction(8, -12)
```

Problem 4. Add test cases to the unit tests provided above to get full coverage for the `__init__()`, `__str__()`, `__float__()`, and `__eq__()` methods. You may modify the fixture function if it helps. Also add unit tests for the magic methods `__add__()`, `__sub__()`, `__mul__()`, and `__truediv__()`. Verify that you have full coverage with `pytest-cov`.

Additionally, **two** of the `Fraction` class's methods are implemented incorrectly. Use your tests to find the issues, then correct the methods so that your tests pass.

See <http://doc.pytest.org/en/latest/index.html> for complete documentation on `pytest`.

Test-driven Development

Test-driven development (TDD) is the programming style of writing tests **before** implementing the actual code. It may sound tedious at first, but TDD incentivizes simple design and implementation, speeds up the actual coding, and gives quantifiable checkpoints for the development process. TDD can be summarized in the following steps:

1. Define with great detail the program specifications. Write function declarations, class definitions, and especially docstrings, determining exactly what each function or class method should accept and return.
2. Write a unit test for each unit of the program, usually black box tests.
3. Implement the program code, making changes until all tests pass.

For adding new features or cleaning existing code, the process is similar.

1. Redefine program specifications to account for planned modifications.
2. Add or modify tests to match the new specifications.
3. Change the code until all tests pass.

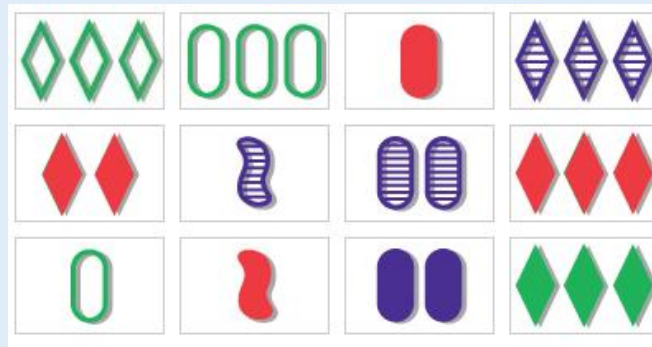


If the test cases are sufficiently thorough, then when the tests all pass, the program can be considered complete. Remember, however, that it is not sufficient to just have tests, but to have tests that accurately and rigorously test the code. To check that the test cases are sufficient, examine the test coverage and add additional tests if necessary.

See https://en.wikipedia.org/wiki/Test-driven_development for more discussion on TDD and https://en.wikipedia.org/wiki/Behavior-driven_development for an overview of Behavior-driven development (BDD), a close relative of TDD.

Problem 5. *Set* is a card game about finding patterns. Each card contains a design with 4 different properties: color (red, green or purple), shape (diamond, oval or squiggly), quantity (one, two, or three) and pattern (solid, striped or outlined). A *set* is a group of three cards which are either all the same or all different for each property. You can try playing *Set* online at <http://smart-games.org/en/set/start>.

Here is a group of twelve Set cards.



This collection of cards contains six unique sets:



(a) Same in quantity and shape; different in pattern and color

(b) Same in color and pattern; different in shape and quantity



(c) Same in pattern; different in shape, quantity and color

(d) Same in shape; different in quantity, pattern and color



(e) Different in all aspects

(f) Different in all aspects

Each Set card can be uniquely represented by a 4-bit integer in base 3,^a where each digit represents a different property and each property has three possible values. A full hand in Set is a group of twelve unique cards, so a hand can be represented by a list of twelve 4-digit integers in base 3. For example, the hand shown above could be represented by the following list.

```
hand1 = ["1022", "1122", "0100", "2021",
         "0010", "2201", "2111", "0020",
         "1102", "0210", "2110", "1020"]
```

The following function definitions provide a framework for partially implementing Set by calculating the number of sets in a given hand.

```

def count_sets(cards):
    """Return the number of sets in the provided Set hand.

    Parameters:
        cards (list(str)) a list of twelve cards as 4-bit integers in
        base 3 as strings, such as ["1022", "1122", ..., "1020"].
    Returns:
        (int) The number of sets in the hand.
    Raises:
        ValueError: if the list does not contain a valid Set hand, meaning
            - there are not exactly 12 cards,
            - the cards are not all unique,
            - one or more cards does not have exactly 4 digits, or
            - one or more cards has a character other than 0, 1, or 2.
    """
    pass

def is_set(a, b, c):
    """Determine if the cards a, b, and c constitute a set.

    Parameters:
        a, b, c (str): string representations of 4-bit integers in base 3.
        For example, "1022", "1122", and "1020" (which is not a set).
    Returns:
        True if a, b, and c form a set, meaning the ith digit of a, b,
        and c are either the same or all different for i=1,2,3,4.
        False if a, b, and c do not form a set.
    """
    pass

```

Write unit tests for these functions, but **do not** implement them yet. Focus on *what* the functions should do rather than on *how* they will be implemented.

(Hint: if three cards form a set, then the first digits of the cards are either all the same or all different. Then the sums of these digits can only be 0, 3, or 6. Thus, a group of cards forms a set only if for each set of digits—first digits, second digits, etc.—the sum is a multiple of 3.)

^aA 4-bit integer in base 3 contains four digits that are either 0, 1 or 2. For example, 0000 and 1201 are 4-bit integers in base 3, whereas 000 is not because it has only three digits, and 0123 is not because it contains the number 3.

Problem 6. After you have written unit tests for the functions in Problem 5, implement the actual functions. If needed, add additional test cases to get full coverage.

(Hint: The `combinations()` function from the standard library module `itertools` may be useful in implementing `count_sets()`.)

Additional Material

The Python Debugger

Python has a built in debugger called `pdb` to aid in finding mistakes in code during execution. The debugger can be run either in a terminal or in a Jupyter Notebook.

A *break point*, set with `pdb.set_trace()`, is a spot where the program pauses execution. Once the program is paused, use the following commands to tell the program what to do next.

Command	Description
<code>n</code>	n ext: executes the next line
<code>p <var></code>	p rint: display the value of the specified variable.
<code>c</code>	c ontinue: stop debugging and run the program normally to the end.
<code>q</code>	q uit: terminate the program.
<code>l</code>	l ist: show several lines of code around the current line.
<code>r</code>	r eturn: return to the end of a subroutine.
<code><Enter></code>	Execute the most recent command again.

For example, suppose we have a long loop where the value of a variable changes unpredictably.

```
# pdb_example.py
import pdb
from random import randint

i = 0
pdb.set_trace()                # Set a break point.
while i < 1000000000:
    i += randint(1, 10)
print("DONE")
```

Run the file in the terminal to begin a debugging session.

```
$ python pdb_example.py
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) l                        # Show where we are.
 2     import pdb
 3     from random import randint
 4
 5     i = 0
 6     pdb.set_trace()
 7 -> while i < 1000000000:
 8         i += randint(1, 10)
 9     print("DONE")
[EOF]
```

We can check the value of the variable `i` at any step with `p i`, and we can even change the value of `i` mid-program.

```

(Pdb) n                                # Execute a few lines.
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) n
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) n
> /Users/Student/pdb_example.py(8)<module>()
-> i += randint(1, 10)
(Pdb) p i                                # Check the value of i.
8
(Pdb) n                                # Execute another line.
> /Users/Student/pdb_example.py(7)<module>()
-> while i < 1000000000:
(Pdb) p i                                # Check i again.
14
(Pdb) i = 999999999                      # Change the value of i.
(Pdb) c                                # Continue the program.
DONE

```

See <https://docs.python.org/3/library/pdb.html> for documentation and examples for the Python debugger.

Other Testing Suites

There are several frameworks other than `pytest` for writing unit tests. Each shares the same basic structure, but the setup, syntax, and particular features vary. For more unit testing practice, try out the standard library's `unittest` (<https://docs.python.org/3/library/unittest.html>) or `doctest` (<https://docs.python.org/3/library/doctest.html>), or the third-party `nose` module (<https://nose.readthedocs.io/en/latest/>). For a much larger list of unit testing tools, see <https://wiki.python.org/moin/PythonTestingToolsTaxonomy>.

The Fractions Module

The standard library's `fractions` module (<https://docs.python.org/3/library/fractions.html>) has a `Fraction` class that is similar to the `Fraction` class presented in this lab. Its structure and syntax is a little different from this lab's class, but it is a little more robust in that it can take in floats, decimals, integers, and strings to its constructor. See also the `decimals` module (<https://docs.python.org/3/library/decimal.html>) for tools relating to decimal arithmetic.