

Lecture 1: Getting started on the command line

Command line

Python is run from the command line. You should become familiar with how to navigate the command line as it is essential for any sort of programming.

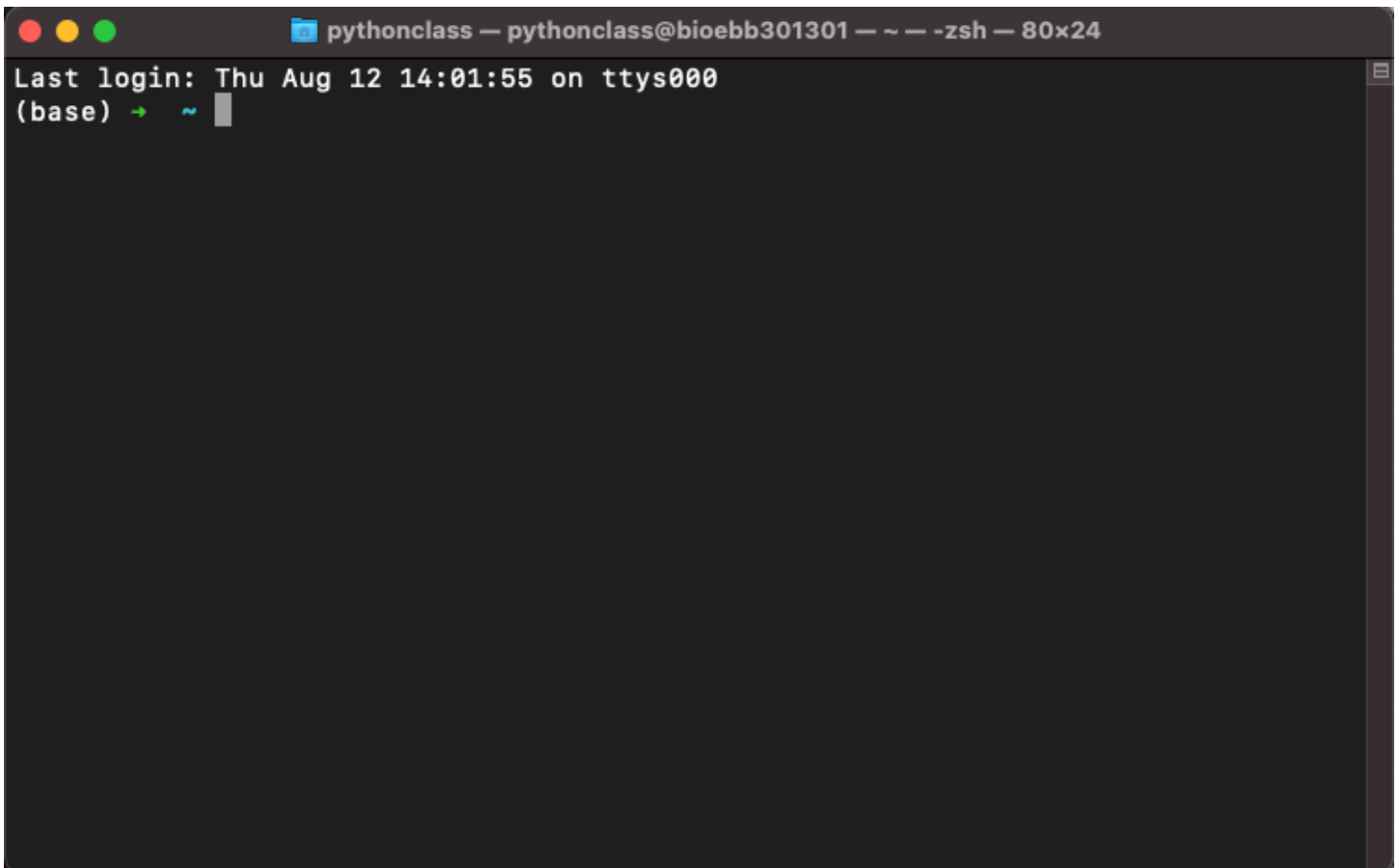
There are two essential ways to interact with a computer. A graphical user interface (GUI) is what you are probably most familiar with, using a mouse, touchpad, or finger to interact with what is on the screen. However, you also can interact with the computer through a command-line interface (CLI). In a CLI, the user types commands that tell the computer to perform desired tasks.

The Unix command line, where Unix refers to a family of operating systems that include Linux, Android, iOS, and macOS. The one exception is the Microsoft Windows system, however, PCs now ship with Linux kernels that mimic a Unix environment (<https://docs.microsoft.com/en-us/windows/wsl/install-win10>).

Running a command

On MacOS, you can open a terminal window by using the Spotlight application (⌘-space), search for Terminal, and run the program. On Linux, you should see the terminal icon on the left hand side of the screen. It will be a black box with a `>_` in the center. On Windows, use the Linux kernel as linked above.

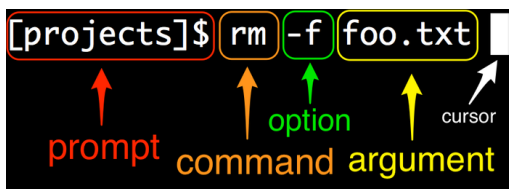
Your terminal will look something like this:

A screenshot of a terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left, followed by the text "pythonclass — pythonclass@bioebb301301 — ~ — -zsh — 80x24". The terminal content shows "Last login: Thu Aug 12 14:01:55 on ttys000" and a prompt "(base) → ~" with a cursor. The background is dark gray.

You can now type into the terminal and issue commands to your computer.

Below is an example of a command you can run in the terminal. There are a few terms you should memorize:

1. **Prompt** - A prompt is supplied automatically by the terminal, and you do not type it in. The prompt gives you information about your current directory, the user, etc. This information is customizable. In the example above, (base) -> ~ is the prompt. (base) gives you information about the conda environment you are in and the ~ tells you you are in the home directory.
2. **Command** - The command tells the computer to 'do' something. In the example below, rm tells the computer to delete something (permanently, so be careful!). A command can either be built into the Unix operating system or a custom program written by someone else (or you!) and downloaded to your computer. A command is similar to a program in a GUI environment, e.g. Microsoft Word or Google Chrome.
3. **Option** - An option can be recognized by the use of the - or the use of two --. -f or --recursive are two examples of using an option to modify how the computer is run. A single "-" is typically followed by a single letter while a double dash "--" is followed by a word. In the example below, the -f tells the rm command to remove the files without prompting for confirmation.
4. **Argument** - An argument is provided to the command to specify how it is run. In order to remove a file, the command needs to know what file you want to remove.



Below are some common commands that are often used

pwd - prints current directory (or node). Tells you where you are. This command doesn't require any arguments.

ls - Lists all the children nodes (files and directories) in the current directory. Use `-l` to get more detailed information. Use `-a` to list ALL files in a directory. This command doesn't require any arguments.

cd - Move to a new directory. Use `cd /` to go to root. Use `cd ~` to go to your home directory. Use `cd ..` to go to the parent directory. This command requires the name of the path to navigate to.

mkdir - Create a new directory. This command requires the name of a directory to create.

rm - Remove a file. Add `rm -r` removes an entire directory. This command requires the name of a directory to delete. Note that the default of this command is very unsafe as the command immediately and permanently executes. To make it a little bit safer, we will modify our `.bashrc` file and add the following line:

mv - Moves a file or directory around. This command requires the name of the file/directory to move as well as the place to move it to.

cp - Copies a file/directory to a new location. This command requires the name of the file/directory to copy as well as the place to copy it to.

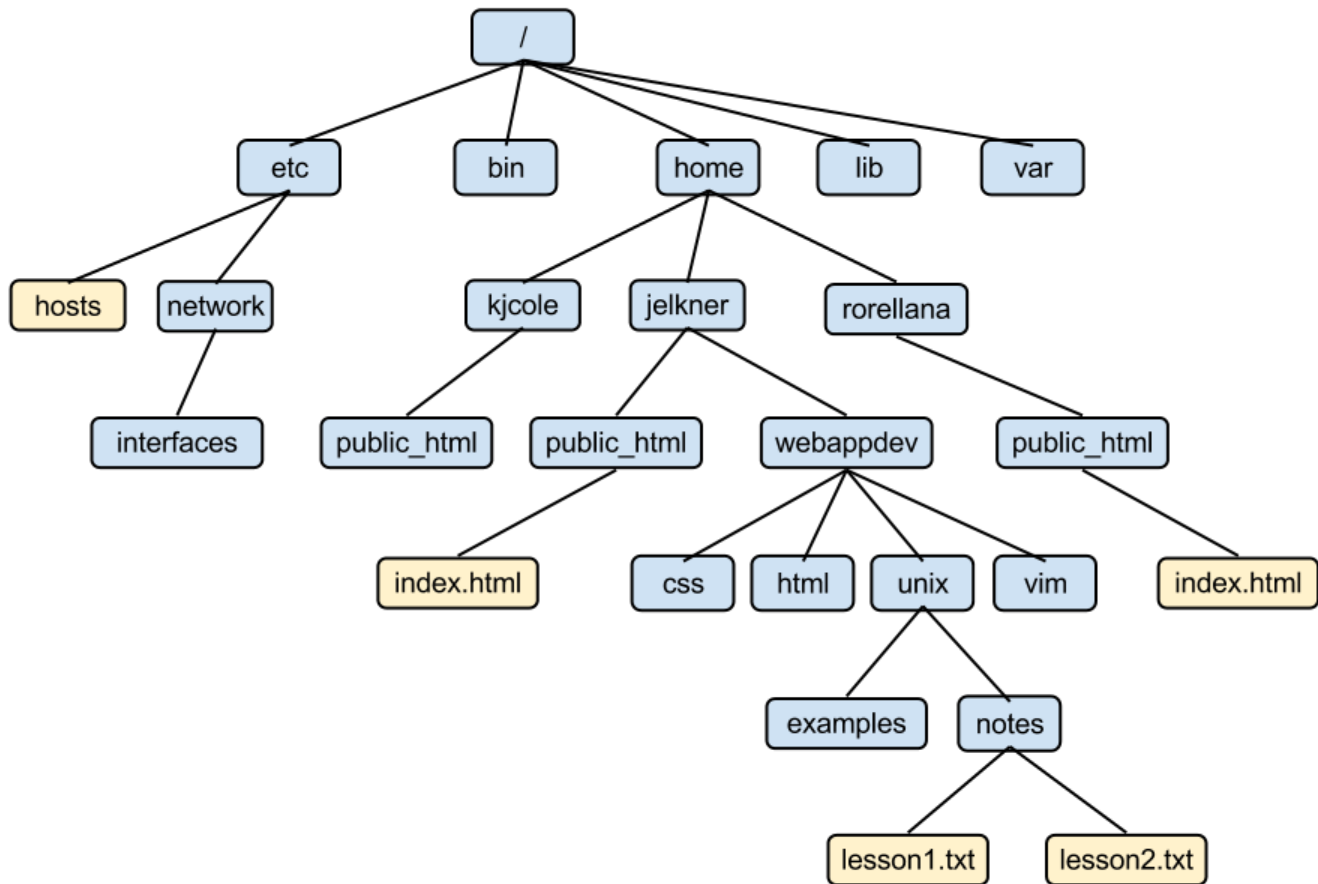
which - This command gives you the absolute path of the executable for a command that you run.

touch - This command creates an empty file with a given name

echo - This command prints something to the screen, either text or the value of something

man - This command prints help on a command

File Directory



The command line interface, lives within a specific directory. At any given moment, you are within a specific directory, which you can change using the `cd` command. The file directory is a tree structure. The top of the directory is called root. You can go to the root directory using the `/`, as so:

```
(base) → ~ cd /
```

```
(base) → / ls
```

Applications	System	bin	etc	private	usr
Library	Users	cores	home	sbin	var
Network	Volumes	dev	opt	tmp	

Note that the prompt changes and now we see the top level directories (or symbolic links to different directories in pink). We can navigate back to the home directory like so:

```
(base) → / cd ~
```

```
(base) → ~ ls
```

Applications	Downloads	Movies	Public
Desktop	Dropbox (GaTech)	Music	opt
Documents	Library	Pictures	

Notice that again the prompt has changed to tell us we are in a different place. You can also navigate to individual directories within a directory:

```
(base) → ~ cd Pictures
```

```
(base) → Pictures ls
Photos Library.photoslibrary
(base) → Pictures cd Photos\ Library.photoslibrary
(base) → Photos Library.photoslibrary
```

Here we have first navigated to the Pictures directory and then we have navigated to a directory within there called "Photos Library.photoslibrary". Notice that the CLI has a hard time with spaces. The name of the directory has a space within it, but the cd command uses whitespace to separate arguments. The solution is to use the '\ ' before the space to indicate that the space is still part of the name.

You should also have a GUI to view your file directory. On a Mac it is called Finder. You should be able to identify all of the directories on both the GUI and the CLI at the same time.

Other useful features

Tab – Tab can be used to automatically complete the name of a file that you are typing. In your home directory, type "cd Libr" and then hit tab. What happens?

Up arrow – Your shell keeps track of all the commands you have run. You can easily access them by hitting the up arrow to scroll through your previous commands. A useful way to combine the tab features (automated completion) and command history is to start typing the command you want to find and then hitting the up arrow. This filters the search history to commands that match what you have started typing. This features is included with oh-my-zsh but you can also add the following to your .bashrc file.

```
bind '"\e[A": history-search-backward'
```

Type bash in your prompt to reload your preferences. What happens if you type cd and then up?

Ctrl-A and Ctrl-E are useful to remember for jumping to the beginning or end of the command you are typing. This is extremely useful for modifying your previous commands (especially as they get longer and longer).

If you want to go to a specific part of a command, you might think you could click on it. However, this does not work. If you hold option while you click on the command, you will be able to click to specific parts of the command.

The asterisk: * - The asterisk represents any number of unknown characters. Use it when searching for documents or files for which you have only partial names.

Different types of paths

There are three types of paths you should be aware of

1. Absolute path – this is when you specify the entire path starting from the root directory
2. Relative path – this is when you specify the path relative to your current directory
3. Lookup path – your shell has the ability to find things based upon an environmental variable called PATH. Environmental variables allow you to customize the shell for your machine. The PATH specifies directories that the shell looks for commands that you commonly run.

Let's see a few ways these work. First Absolute path: let's navigate to the Documents folder in my home directory using an absolute path

```
(base) → ~ pwd
/Users/pythonclass
(base) → ~ ls
Applications      Downloads      Movies      Public
Desktop           Dropbox (GaTech) Music      opt
Documents         Library      Pictures
(base) → ~ cd /Users/pythonclass/Documents
(base) → Documents ls
(base) → Documents cd /Users/pythonclass/Documents
```

```
(base) → Documents cd /  
(base) → / cd /Users/pythonclass/Documents  
(base) → Documents
```

I first used the `pwd` command to determine the absolute path of my current directory. I then used this information to `cd` into Documents starting from root. Note that I started the path with the `/`. You can also use the `~` sign to indicate you are starting from your home directory. This tells the shell that I want to use an absolute path. Note that I can run the `cd` command over and over again, no matter what directory I start from, and I get to the same location.

Now let's try relative paths:

```
(base) → Documents cd ~  
(base) → ~ cd Documents  
(base) → Documents cd Documents  
cd: no such file or directory: Documents  
(base) → Documents
```

First, I navigated back to my home directory using an absolute path (the `~` is shorthand for `/Users/pythonclass/`), then I used the relative path to go into the Documents. Note that if I try to run that command again I get an error: that's because the Documents folder does not have a Documents folder within it. If you don't specify the `/` at the start of the path, you are using a relative path.

Finally, relative paths are used to find executables. Executables are programs or scripts that can do something. We haven't encountered any executable files yet but `python3` is actually an example of an executable program. Let's find it in our miniconda installation. You specified the location of this during your installation. Typically it is installed in your home directory in `~/miniconda` or `~/opt/miniconda`.

```
(base) → ~ cd opt/miniconda3/bin  
(base) → bin ls
```

2to3	lzdiff	python.app	unlzma
2to3-3.9	lzegrep	python3	unxz
activate	lzfgrep	python3-config	wheel
c_rehash	lzgrep	python3.9	wish
captoinfo	lzless	python3.9-config	wish8.6
chardetect	lzma	pythonw	xz
clear	lzmadec	reset	xzcat
conda	lzmainfo	sqlite3	xzcmp
conda-env	lzmore	sqlite3_analyzer	xzdec
cph	ncursesw6-config	tabs	xzdiff
deactivate	openssl	tclsh	xzegrep
idle3	pip	tclsh8.6	xzfgrep
idle3.9	pip3	tic	xzgrep
infocmp	pydoc	toe	xzless
infotocap	pydoc3	tput	xzmore
lzcat	pydoc3.9	tqdm	
lzcmp	python	tset	

Notice there are two colors files within this directory. Orange files are executables. The `bin` directory is generally used to hold executable files. Pink files are symbolic links to files or directories. Essentially this allows you to rename an executable with less verbose names. You can see what these link to by using the `-la` flags on the `ls` command:

```
(base) → bin ls -la py*  
lrwxr-xr-x  1 pythonclass  staff    8 Aug 12 13:26 pydoc -> pydoc3.9
```

```

lrwxr-xr-x  1 pythonclass  staff          8 Aug 12 13:26 pydoc3 -> pydoc3.9
-rwxrwxr-x  1 pythonclass  staff       107 Aug 12 13:26 pydoc3.9
lrwxr-xr-x  1 pythonclass  staff          9 Aug 12 13:26 python -> python3.9
-rwxrwxr-x  1 pythonclass  staff      153 Aug 12 13:26 python.app
lrwxr-xr-x  1 pythonclass  staff          9 Aug 12 13:26 python3 -> python3.9
lrwxr-xr-x  1 pythonclass  staff        16 Aug 12 13:26 python3-config ->
python3.9-config
-rwxrwxr-x  1 pythonclass  staff    4643600 Aug 12 13:26 python3.9
-rwxrwxr-x  1 pythonclass  staff      2066 Aug 12 13:26 python3.9-config
-rwxrwxr-x  1 pythonclass  staff      153 Aug 12 13:26 pythonw

```

The `py*` argument restricts the files that are listed to those that start with `py`

`python3` is a symbolic link to the `python3.9` executable (orange) while `python3-config` is a symbolic link to the `python3.9-config` executable.

We can run the `python3.9` executable in a few ways:

```

(base) → bin ./python3.9
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
(base) → bin ./python3
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
(base) → bin ./python
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

```

We use the `./` to specify that we want to run the executable file in the current directory. In the first case we used the executable file itself to start the `python3` interpreter. In the second two cases we used the symbolic links to start the interpreter.

Now let's go up a directory and try to run `python3`:

```

(base) → bin cd ..
(base) → miniconda3 ls
LICENSE.txt  conda-meta  etc         lib          python.app  shell
bin          condabin   include     pkgs         share       ssl
(base) → miniconda3 python3
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
(base) → miniconda3 python3.9
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

```

If you aren't familiar with CLIs, this might seem surprising. We are in a directory where this is no executable, but we can still start the `python` interpreter using the name of the executable or the symbolic links. How is this possible? This is possible because the shell has a few defined directories that it looks into for executables to

run. These directories are found in the PATH environmental variable which you can see using the echo command (Use a \$ to specify that you want to know the value of an environmental):

```
(base) → miniconda3 echo $PATH
/Users/pythonclass/opt/miniconda3/bin:/Users/pythonclass/opt/miniconda3/condabin:
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/salt/bin:/usr/local/sbin:/usr/l
ocal/munki:/usr/local/mysql/bin:/opt/X11/bin
```

Using a little bit of unix magic, we can list each directory separately per line:

```
(base) → miniconda3 echo $PATH | tr ':' '\n'
/Users/pythonclass/opt/miniconda3/bin
/Users/pythonclass/opt/miniconda3/condabin
/usr/local/bin
/usr/bin
/bin
/usr/sbin
/sbin
/opt/salt/bin
/usr/local/sbin
/usr/local/munki
/usr/local/mysql/bin
/opt/X11/bin
```

On my computer, there are 12 directories that the shell looks for executables, the first being the location of the conda installation of the python interpreter. Your computer probably has a different number of directories.

If you want to see where the executable is found, you can use the which command:

```
(base) → miniconda3 which python3
/Users/pythonclass/opt/miniconda3/bin/python3
```

This tells me that when I type python3, I am really running: /Users/pythonclass/opt/miniconda3/bin/python3.

```
(base) → miniconda3 /Users/pythonclass/opt/miniconda3/bin/python3
Python 3.9.5 (default, May 18 2021, 12:31:01)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note that the order of the directories in the path is the order that the shell looks for a command. For example, there is also an installation of python3 in the /usr/bin and /usr/local/bin/ directories:

```
(base) → miniconda3 /usr/bin/python3
Python 3.8.2 (default, Apr 8 2021, 23:19:18)
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
(base) → bin /usr/local/bin/python3
Python 3.9.6 (default, Jun 29 2021, 05:25:02)
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

We can change the PATH variable to change the order of directories that we look in. export is a command that can be used to change an environmental variable.

```
(base) → bin echo $PATH
/Users/pythonclass/opt/miniconda3/bin:/Users/pythonclass/opt/miniconda3/condabin:
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/salt/bin:/usr/local/sbin:/usr/l
ocal/munki:/usr/local/mysql/bin:/opt/X11/bin
```

```
(base) → bin export
PATH=/Users/pythonclass/opt/miniconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr
/sbin:/sbin:/opt/salt/bin:/usr/local/sbin:/usr/local/munki:/usr/local/mysql/bin:/
opt/X11/bin:/Users/pythonclass/opt/miniconda3/bin
(base) → bin echo $PATH
/Users/pythonclass/opt/miniconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/sbin
:/sbin:/opt/salt/bin:/usr/local/sbin:/usr/local/munki:/usr/local/mysql/bin:/opt/X
11/bin:/Users/pythonclass/opt/miniconda3/bin
```

Note that the miniconda installation is now the last directory in PATH instead of the 1st. What happens if we run Python3 now?

```
(base) → bin python3
Python 3.9.6 (default, Jun 29 2021, 05:25:02)
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
(base) → bin which python3
/usr/local/bin/python3
```

Now we are using the python3 executable found in /usr/local/bin, which is not from the miniconda installation. If we quit and reload the terminal, you will revert back to the default order, which is specified in the configuration file for your shell.

There are a number of other environmental variables. A few common ones:

```
(base) → bin echo $HOME
/Users/pythonclass
(base) → bin echo $USER
pythonclass
(base) → bin echo $SHELL
/bin/zsh
(base) → bin echo $PWD
/usr/local/bin
(base) → bin cd
(base) → ~ echo $PWD
/Users/pythonclass
(base) → ~ echo $LANG
en_US.UTF-8
```