Concepts introduced: command line (terminal and shell), remote computing, textfiles and editors, Python, extensible scripts.

1 Introduction to Command Line

A command line interface (CLI) is a convenient and powerful way to interact with a computer. It often takes a bit of adjustment for a person who is used to graphical user interfaces (GUI) to get up and running with CLIs. However, the investment is *always* worth it. CLIs make repetition and automation quite simple. It is much easier to send your colleague a shell command to achieve a task compared to a sequence of GUI instructions.

Note In all documentation for CME211 the dollar sign symbol (`$`) will be used to indicate a shell command. All shell commands in these notes (and all CME211 material) are geared for `bash`, but will likely work in `tcsh`. The pound symbol (`#`) is used to indicate shell comments. Inline, you might see something like "try the command `$ pwd`". Code blocks (like the following) will also be extensively used for demonstration. Note that you don't actually type the `$` before the command.

```
# This is a comment, the $ on the next line is followed by a command.
$ pwd
/Users/andreas_santucci/Dropbox/
# the previous line was output from the pwd command
```

1.1 Terminal

A *terminal*, *terminal emulator*, or *console* is a program that displays text and handles input. These programs emulate the behavior of physical computer terminals (also known as dumb terminals) in past computing systems. Users of modern computing systems often have many terminal windows open at once. In the past, users were limited to the physical terminal they sat behind.

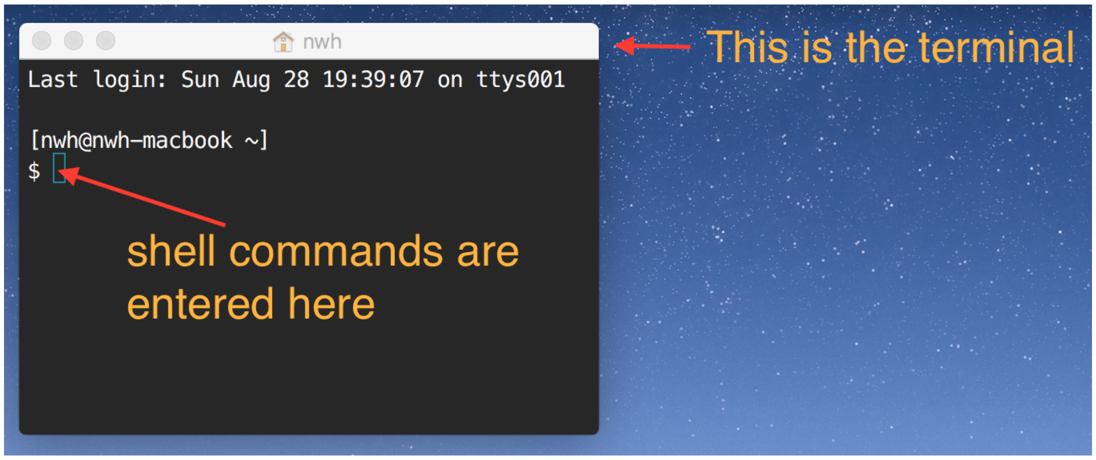
Mac On macOS, the built-in terminal program is called `Terminal.app`. It is located in `/Applications/Utilities`. One convenient way to start the program is to search for `terminal` using Spotlight.

Windows On Windows operating systems, the built-in terminal program is called the `Command Prompt`. You may access it by navigating through `Start -> All Programs -> Accessories -> Command Prompt`.

Unix If you're on a Unix operating system already, you likely know how to open a terminal. In Ubuntu, for example, you can simply use the keyboard shortcut `Ctrl - Alt + T`.

1.2 Shell

A *shell* is a program that executes commands from the user and displays the result. There are many different shell programs out there. E.g. `bash` is quite popular; it's been around since 1989 and is the default on macOS and most Linux distributions. For a time, `tcsh` was the default *shell* on Farmshare systems. It is possible to change the login shell with the `chsh` command.



```
Last login: Sun Aug 28 19:39:07 on ttys001

[nwh@nwh-macbook ~]
$
```

Windows If you're on Windows 10, I recommend going ahead and installing Ubuntu on your machine. This will allow you to access all of the Unix development tools that we teach.[1] If you don't have Windows 10, you can try something like cygwin or mingw. The easiest route is to simply use secure shell to access a remote resource, see section 3.3.

2 Navigating a Command Line Interface

A *path* specifies the location of a file or directory in a file system hierarchy. On unix-like systems (e.g. macOS and Linux) a single slash (`/`) indicates the very top (or root) of the file system. In longer path names, directories are separated by slashes. The last item (lacking a slash) may be either a file or a directory. If a path ends with a slash, it's a directory.

- `/Users/asantucci/Downloads`: this is the downloads directory on my Mac.
- `/Users/asantucci/Downloads/`: this is also the downloads directory on my Mac. Note the trailing slash to indicate that `Downloads` is a directory.
- `/Users/asantucci/Downloads/syllabus.pdf`: this is the path to a downloaded PDF.
- `~/Downloads`: this is the downloads directory on my Ubuntu machine.

[1] While the Command Prompt offered in Windows is still a terminal, it doesn't subscribe to the Unix tool-chain that we teach. Attempts to use something like `Powershell` will not be amenable to our class as the commands are different and don't directly translate from one tool to another.

Shell commands are executed relative to a *working directory*. Usually, when a shell first starts, the working directory is the user's home directory.

- `pwd` - print working directory
- `cd` - change directory

Special directory aliases:

- `~` - user's home directory
- `..` - directory one higher in filesystem
- `.` - alias for working directory

The command `$ cd -` changes to the previous directory.

Avoid spaces in directory and file names It is best to not use spaces in directory or file names. Most shell programs use a space as a delimiter between commands and arguments. Thus, spaces in file or directory names need to be escaped or quoted – a thing that is easy to forget.

For example, let's say we have a directory called `my docs`. If we try to enter the directory with `cd` with out handling the space, we get an error:

```
$ cd my docs
-bash: cd: my: No such file or directory
```

To make this work, we can either quote the directory name (note the "funny looking" underscore in the below text, and in these notes, is actually syntactically denoting a space):

```
$ cd "my␣docs"
```

Or escape the space with a backslash:

```
$ cd my\ docs
```

2.1 Looking at things

- `ls` - list files (and folders) in a directory
- `cat` - dump file contents to terminal output
- `less` - open file in a "pager" (hit 'q' to quit!)
- `file` - inspect file type and print helpful information about file

2.2 Manipulating files

- `cp` - copy files and directories
- `mv` - move or rename files and directories
- `rm` - remove files and directories (**be careful:** files cannot be recovered after `rm`)
- `touch` - create file or update timestamp
- `mkdir` - create directories

2.3 Inspecting commands

- `type` - Display information about the command type (useful for aliased commands)
- `which` - Locate a command (i.e. find where the executable is located)
- `help` - Display reference page for shell builtin
- `man` - Display an on-line command reference (hit ‘q‘ to quit)

2.4 Quitting a command

Sometimes you need to terminate a command. This is often possible with the `ctrl-c` keyboard command. In documentation you might see this represented as `^C`, where `^` is a symbol indicating the ‘ctrl‘ key.

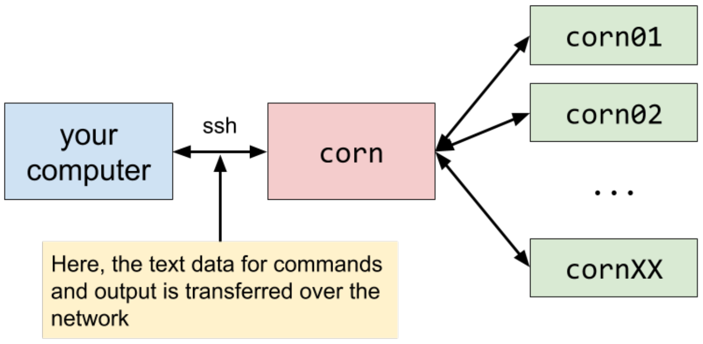
Resources

- http://linuxcommand.org/
- http://www.pixelbeat.org/cmdline.html
- http://software-carpentry.org/
- http://swcarpentry.github.io/shell-novice/
- https://www.youtube.com/watch?v=hAHJ0xGKMBk

3 Remote Computing

Often various resources (data, programs, high performance computing) are located somewhere other than the computer we have in front of us. There are many tools and methods for accessing remote computing resources. In CME211, we will use `ssh` to access the shared computing resources managed by Stanford Research Computing. In particular, we will be using the `rice` servers on the Farmshare System.

Structure of Rice The `rice` servers are accessed through the address `rice.stanford.edu`. The “master node” is called `rice`‘. This will send users to one of the worker nodes, designated `riceXX` where `XX` is a number.[2]


your computer
ssh
corn
corn01
corn02
...
cornXX
Here, the text data for commands and output is transferred over the network

Logging into a Remote Computer `ssh` stands for “secure shell”. The program encrypts the communication between client and server. The command to log into ‘rice‘ via ssh is:

```
$ ssh [stanford_username]@rice.stanford.edu
```

Here, `stanford_username` needs to be replaced by your username. SSH will then attempt to locate the server, authenticate the user, and provide access to a shell.

Here is the terminal output when I log into ‘rice.stanford.edu‘.

```
$ ssh santucci@rice.stanford.edu
Warning: Permanently added the RSA host key for IP address
         '171.67.216.71' to the list of known hosts.

santucci@rice.stanford.edu's password:
Authenticated with partial success.
Duo two-factor login for santucci
Enter a passcode or select one of the following options:

1. Duo Push to XXX-XXX-XXXX
2. Phone call to XXX-XXX-XXXX
3. SMS passcodes to XXX-XXX-XXXX

Passcode or option (1-3): 1
Success. Logging you in...
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-33-generic x86_64)
# ... many lines omitted ...
For questions or concerns, please contact:
 research-computing-support@stanford.edu

[santucci@rice06 ~] $ pwd
/home/santucci
```

For more information on SSH, see `$ man ssh`, where `man` is the “manual” or “help” command, useful for fetching documentation for a particular command.

[2]We previously used a cluster call `corn`, whence the text within the graphic presented is outdated.

3.1 Transfering Files (to and from `rice`) using `scp`

The `scp` (stands for "secure copy") tool can be used to copy files to and from a remote computer running an SSH server. The following command will copy from `source_file` to `dest_file`:

```
$ scp source_file dest_file
```

If one (or both) of the files is on a remote computer, then the user and server address must be specified. For example, I could copy the file `demo/doggo.txt` to my home directory on `rice` with the following command:

```
$ scp demo/doggo.txt santucci@rice.stanford.edu:~/
# authentication
doggo.txt                                    100%  187     0.2KB/s
00:00
```

Note the colon (`:`) between the server name and path in the above `scp` command.

Now looking on `rice`:

```
[santucci@rice06 ~]
$ ls -1
afs-home
doggo.txt
```

Going the other way is simple as well, i.e. copying a file from Rice to your local computer; we simply execute the following command *from* our local machine to download a file:

```
scp santucci@rice.stanford.edu:~/doggo.txt .
```

3.2 Farmshare user directory

The Farmshare remote computing resource offers another location for users to store files known as the "Farmshare user directory". This is located at `/farmshare/user_data/[sunet_id]`. For example, my `sunet_id` is `santucci`, therefore my Farmshare user directory is `/farmshare/user_data/santucci`.

Some important notes from the Farmshare User Guide:

- Your Farmshare user directory will likely **not exist** the first time you login. A script will run and notice your login and create the directory within about 30 minutes of your first Farmshare login.
- `/farmshare/user_data` is **NOT** backed up. Make sure your push your work to GitHub on a regular basis. You will learn how to do this in HW0.

- Space is limited on `/farmshare`. There currently is no quota system. Keep your Farmshare user directory under `5GB`.

You can `scp` files directly to your Farmshare user directory. For example (from the local `lecture-00` directory):

```
$ echo $HOSTNAME
santucci-mbpro.local     # this is my laptop
$ pwd
/Users/santucci/git/cme211-notes/lecture-00
$ scp demo/doggo.jpg santucci@rice.stanford.edu:/farmshare/user_data/santucci/
# authentication
doggo.jpg                                    100%   770KB 770.3KB/s    00:00
$ ssh santucci@rice.stanford.edu
# authentication
# now logged into rice.stanford.edu
$ echo $HOSTNAME
rice23.stanford.edu
$ cd /farmshare/user_data/santucci/
$ ls
doggo.jpg
```

3.3 Other Tools

There are numerous software tools for working with files on remote computing systems.

There web utilities, ***accessible from any OS***, e.g. Stanford WebAFS provides a web interface to files on Farmshare, and Google has a beta Chrome-based SSH client.

Here is a list of ***Windows specific*** GUI based tools to for accessing remote computers:

SecureCRT, SecureFX, PuTTY, and Bitvise SSH client.

The ***Mac OS*** comes with `ssh` and `scp`. The program Fetch provides a GUI for file transfer: . The `rsync` command may be used to sync an entire directory:

```
$ rsync -avz -e ssh remoteuser@remotehost:/remote/dir /this/dir/
```

Search for rsync over ssh for more information on this.

4 Text Files

A *text file* is simply a sequence of characters that can be opened by any *text editor*. In scientific computing and data science applications, text files can be used for a variety of tasks:

- Publication and presentation with LaTeX (`.tex` file extension)
- Dataset storage in comma-separated value files (`.csv`)

- Object serialization with JSON (`.json`)
- Websites (`.html`) or markdown files (`md`).
- Graphics with tools like SVG or TikZ.
- Source code in any language, CME211 will use Python (`.py`) and C++ (`.cpp`)
- Tools for 3d models.
- Software build systems with GNU Make or CMake.
- Music notation

One benefit of working with text files is that they can be checked into version control systems and easily compared to previous versions. It is also possible to check binary files into a version control system, but it is not as easy to find the differences from previous versions; some examples of binary files include `.zip`, `.jpg`, `.xls`, `.docx`.

4.1 Text Editors

Text editors that work in a terminal include (my personal favorite) Emacs, Vim, and Nano.

GUI Based text editors include Atom, Sublime Text, Visual Studio Code, and TextWrangler.[3]

5 Interacting with Python

- Python is a high level language that typically runs in an *interpreter*.[4]
- An *interpreter* is a program that executes statements from a high level language.
- Examples of high level interpreted languages: Python, R, Matlab, Perl, JavaScript
- The most widely used Python interpreter (or implementation) is called **CPython**. It is written in C. There are others, for example **Jython** and **IronPython**. It is fairly easy (with experience) to access code written in C from **CPython**.

  There's even Pypy, an implementation of Python written in Python.
- Python now has a long history. It started in '89 as a project to keep Guido van Rossum occupied during the week around Christmas; version 1.0 was released in 1994.
- This class will use Python 3, which has important differences from Python2.

[3] **Caution:** you likely will feel more comfortable using a GUI based text editor *at first*. However, I strongly recommend learning to at least feel comfortable working out of a terminal editor: they are guaranteed to be available on any Unix system, and they have existed for decades; the "features" that come with paid versions of GUI counterparts often include functionality that is often decades old and freely available if you're comfortable learning to read documentation. If you want something more modern, consider xi.

[4] You may have heard the expression low level language, which allow the programmer to interface with the computer's instruction set architecture; this means dealing with registers, memory addresses, and call stacks. A high level language, in contrast, allows the programmer to think instead about variables, arrays, subroutines. Higher level languages focus on usability over efficiency.

5.1 Getting started

Let's log into `rice.stanford.edu`, start the Python 3 interpreter, and execute Python code.

1. Use SSH to login with `ssh [sunet_id]@rice.stanford.edu`
2. Run the Python 3 interpreter with `$ python3`
3. Execute the python statement `>>> print("Hello World!")`.

5.1.1 Interpreter

- An *interpreter* is a program that reads and executes commands.
- It is also sometimes called a REPL or read-evaluate-print-loop.
- One way to interact with Python is to use the interpreter.
- This is useful for interactive work, learning, and simple testing.
- When you see a `$` in code blocks, it typically indicates a shell command. For example:

  ```
  $ ls -1 *.md
  0-outline.md
  1-values-variables-types.md
  2-strings.md
  3-numbers.md
  ```

- A `>>>` in code blocks signifies a command for the Python interpreter.
- The basic Python interpreter is good for simple computations or checks. IPython provides more functionality (e.g. tab completion, syntax highlighting); see `$ ipython3`.

5.1.2 Python as a calculator

In the Python 3 interpreter:

```
>>> 4+7
11
>>> 55*2
110
>>> 9-1.4
7.6
# Next, we show differences between
# (floor) division
>>> 5/3
1.6666666666666667
>>> 5//3
1
>>> -5//3
-2
>>> 5.0/3
1.6666666666666667
>>> 5.0//3
1.0
```

```
>>> 5 % 3                                   2
```

Surprised?

- Division between two integers with `/` returns a floating point number.
- The operator `//` performs floor division (rounds down).
- The `%` (modulus) operator returns the remainder for integer division.

5.1.3 Integers and floating point

We'll discuss details of computer representation of numeric values later, but for now:

- It is best to think of integers as being represented exactly over a fixed range.[5]
- Floating point numbers are *approximations* of real numbers over a limited range.
- Floating point number range is not continuous: the size of gaps between adjacent floating point numbers depend on the scale.[6]
- These things matter; bad numerical computing has resulted in disasters.

5.1.4 Exiting the interpreter

We simply use either `ctrl-d` or `exit()`

```
$ python3
Python 3.5.2 (default, Jun 29 2016, 13:43:58)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> exit
Use exit() or Ctrl-D (i.e. EOF) to exit
>>> exit()
```

5.1.5 Python scripts

- A more convenient way to interact with Python is to write a script
- A Python script is a text file containing Python code
- Python script file names typically end in `.py`

[5]This is not really true in current versions of Python, but will be true in C++.

[6]The gap between `1.0` and the next representable floating point number is smaller than the gap between `1.0e50` and the next representable floating point number.

Let's create our first script:

- Log into `rice.stanford.edu`
- Create a text file named `firstscript.py` with your favorite text editor.

  (`$ nano firstscript.py` is a good choice)
- Insert the following Python code into `firstscript.py`:

```python
print("Hello from Python")
print("I am your first script!")
```

- Execute the command `$ python3 firstscript.py`

Note the use of the `$ python3` command. On many systems the command `$ python` will start the Python 2 interpreter. For this simple example, the behavior will be the same. In general, this is not the case Python versions 2 and 3 have many differences.

5.2 Scripts are Extensible

Let's write a simple Python script to compute the first `n` numbers in the Fibonacci series. As a reminder, each number in the Fibonacci series is the sum of the two previous numbers. Let $F(i)$ be the ith number in the series. We define $F(0) = 0$ and $F(1) = 1$, and $F(i) = F(i-1) + F(i-2)$ for all $i \geq 2$. Numbers `F(0)` to `F(n)` can be computed with the following Python code:

```python
n = 10                      # We wish to calculate up to Fib(10).

if n >= 0:
    fn2 = 0                 # Initialize f(n-2).
    print(fn2,end=',')
if n >= 1:
    fn1 = 1                 # Initialize f(n-1).
    print(fn1,end=',')
for i in range(2,n+1):      # Here is where we iterate from 2:n.
    fn = fn1 + fn2          # Recurrence relation is f(n) = f(n-1) + f(n-2).
    print(fn,end=',')
    fn2 = fn1               # Update our values...
    fn1 = fn                # ...for both f(n-2) and f(n-1).
print()
```

Note, the above code is a preview of Python syntax that we will review in this course. Now, paste this code into a file named `fib.py`.

Execute the file with the command
`$ python3 fib.py`. The result should like:

```
$ python3 fib.py
0,1,1,2,3,5,8,13,21,34,55,
```

To see the utility of scripts, we need to add a bit more code. Change the first line of `fib.py` to be:

```python
import sys
n = int(sys.argv[1])
```

This is done in the script `fib_extensible.py`, and it instructs the script to obtain the value of `n` from the command line:

```
$ python3 fib_extensible.py 0
0,

$ python3 fib_extensible.py 5
0,1,1,2,3,5,

$ python3 fib_extensible.py 21
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,
```

We have increased the utility of our program by making it simple to run from the command line with different input arguments. CME211 homeworks will work like this; in fact, in HW0 we will outline a simple script which will serve as a basis for which you can build more complex scripts which accept arguments from the command line. This allows your program(s) to be tested on unseen test cases!

5.2.1 Python modules

If you are familiar with MATLAB or R, you may come to Python and be confused by:

```python
sqrt(3)  # --> Yields a NameError: name 'sqrt' not defined.
```

The Python language does not have a built in `sqrt` function; this subroutine exists in the `math` module.

```python
import math
math.sqrt(9)
```

About Python modules:

- A module is a collection of Python resources (functions, variables, objects, classes) that can be easily loaded into Python via `import` statements.
- Modules allow for easy code reuse and organization.

- Modules allow the programmer to keep various functionality in different namespaces.
- There are a large number of modules in the Python Standard Library: https://docs.python.org/3/library/index.html
- It is often useful to explore the Python documentation in the interpreter. See `>>> help(math)` and `>>> help(math.sqrt)` from the interpreter.

5.2.2 Printing

The Python interpreter will echo the output of the last (non-assignment) statement in a code block:

```
1+1   # Echos --> 2.
5+5   # Echos --> 10.
```

```
myvar = 101   # Nothing is printed to console.
```

You can use the `print()` function if you wish to inspect its contents.

```
a = 99
print(a)     # Echos --> 99.
```

By default, `print()` adds a new line character at the end for printing.

```
print("hi")               # These contents are ...
print("cme211")           # ... printed on two lines.
```

This behavior can be changed by setting the `end` keyword parameter in the print function.

```
print("hi", end = ' ')   # Now prints a space after "hi" instead of a newline ('\n')
print("cme211")          # The result looks like -- "hi cme211" on its own line.
```

The `print()` function can print several strings at once on the same line:

```
print("apple", "bananna", "orange")
```

The default separator is a space. This can be changed by setting the `sep` keyword parameter:

```
# Add a comma in addition to a space.
print("apple", "bananna", "orange", sep = ", ")
```

Python strings can be “formatted” with the `format` method:

```
r = 10
```

```python
print("The area of a circle of radius {} is {}".format(r, math.pi * math.pow(r, 2))
```

The curly braces ({}) get replaced by the arguments to `format()` in order.

5.3 Values, types, and variables

5.3.1 Values

A value is the fundamental thing that a program manipulates or uses to perform operations. A value is data.

Here is a string value.

```python
"Hello, world!"
```

Here are some numeric values.

```python
42
12.34
```

Here the *only two* Boolean values.

```python
True
False
```

Values have types associated with them.

5.3.2 Types

In Python there are several fundamental data types:

- `bool`: with values `True` and `False`
- `str`: for strings like `"Hello world"`
- `int`: for integers like `1`, `42`, and `-5`
- `float`: for floating point numbers like `96.8`

Python has a `type()` function to determine the type of a value.

```python
  type(55)               # <class 'int'>
  type(-101.5)           # <class 'float'>
  type(False)            # <class 'bool'>
  type("Hi there")       # <class 'str'>
```

In many programming languages, the terms `class` and `type` are either synonymous or at least closely related.

5.3.3 Variables

One of the most basic and powerful concepts in programming is that of a variable, which associates a name to a value.

```python
message = "hello world!"
n = 42
e = 2.71
print(n)  # Echos 42.
```

The last expression shows its possible to print a variable. Everything after the # symbol is part of a comment that is disregarded by the interpreter.

It is almost always preferred to use variables over values. Why? Easier to update code Easier to understand code (useful naming) For example, What does the following code do:

```python
4.2 * 3.5
```

Yeah, we're multiplying two scalar values, but what's the purpose of the computation? If the values are assigned to variables with meaningful names, we might have something like the following.

```python
length = 4.2
height = 3.5
area = length * height
print(area)
```

Now a person reading the code has a good idea of what the values represent and what the output of the code means.

5.3.4 Variable naming

The name associated with a variable is referred to as an *identifier* Variables names must start with a letter or an underscore, such as

```python
_underscore
underscore
```

The remainder of your variable name may consist of letters, numbers and underscores

```python
password1 = "..."
n00b = math.pi
```

```python
under_scores = "__"
```

Names are case sensitive, i.e. `case_sensitive`, `CASE_SENSITIVE`, and `Case_Sensitive` are all different.

5.3.5 Variable naming style

One letter characters such as `a`, `b`, and `c` are too short and not at all descriptive (in general) for meaningful variable names; I've had coworkers do this and it's difficult for them to debug their own logic errors.

On the other hand, something like `number_of_particles_in_target_region` is too long; I've also had coworkers do this and it leads to simple expressions looking needlessly complex.

A better balance might be `num_target_particles`. Perhaps instead of underscore, we prefer camel case i.e. `numTargetParticles`. This is quite important for code readability. People think about this a lot. See: naming conventions in Python.

5.3.6 Important: don't override built-in names

```python
print(abs(-7))
abs = "Must code"
print(abs(-4))       # TypeError: 'str' object is not callable.
```

5.3.7 Exercise: mortgage calculator

Let's write a simple mortgage calculator to compute the monthly payment for a mortgage.

Here are the parameters:

- L: initial principal or value of the loan
- r: yearly interest rate
- $c = r/12$: monthly interest rate
- n: term of the mortgage in terms of number of months (30 years 360 months)

The formula to compute the fixed monthly payment is, the derivation of which can be found here.

$$P = L\frac{c(1+c)^n}{(1+c)^n - 1}$$

Let's code this in Python:

```python
# mortgage amount in dollars
L = 100000
# yearly interest rate of 4%
r = .04
c = r / 12
# duration of mortgage in months (30 years * 12 months)
n = 360
P = L * (c*(1+c)**n) / ((1+c)**n-1)
print("P = " + str(P))
```

Note that in Python the notation `x**y` is used to compute "`x` raised to the power `y`".

5.3.8 Exercise for you: reverse mortgage calculator

Now, write python code to compute the value of the loan given the monthly payment. The formula is:

$$L = P\frac{(1+c)^n - 1}{c(1+c)^n}$$

6 Reading

From **The Linux Command Line** by William Shotts:

- `http://linuxcommand.org/lc3_learning_the_shell.php#contents`
- Read section 1, 2, 3, 5, and 6 (skip sections 4, 7 and above unless interested)