Learn PYTHON the HARD WAY

THIRD EDITION

A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code

ZED SHAW

LEARN PYTHON THE HARD WAY

A Very Simple Introduction
to the Terrifyingly Beautiful World
of Computers and Code

Third Edition

Zed A. Shaw

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

This page intentionally left blank

Preface

This simple book is meant to get you started in programming. The title says it's the hard way to learn to write code, but it's actually not. It's only the "hard" way because it uses a technique called *instruction*. Instruction is where I tell you to do a sequence of controlled exercises designed to build a skill through repetition. This technique works very well with beginners who know nothing and need to acquire basic skills before they can understand more complex topics. It's used in everything from martial arts to music to even basic math and reading skills.

This book instructs you in Python by slowly building and establishing skills through techniques like practice and memorization, then applying them to increasingly difficult problems. By the end of the book, you will have the tools needed to begin learning more complex programming topics. I like to tell people that my book gives you your "programming black belt." What this means is that you know the basics well enough to now start learning programming.

If you work hard, take your time, and build these skills, you will learn to code.

Acknowledgments

I would like to thank Angela for helping me with the first two versions of this book. Without her, I probably wouldn't have bothered to finish it at all. She did the copy editing of the first draft and supported me immensely while I wrote it.

I'd also like to thank Greg Newman for doing the cover art for the first two editions, Brian Shumate for early website designs, and all the people who read previous editions of this book and took the time to send me feedback and corrections.

Thank you.

The Hard Way Is Easier

With the help of this book, you will do the incredibly simple things that all programmers do to learn a programming language:

1. Go through each exercise.
2. Type in each sample *exactly*.
3. Make it run.

That's it. This will be *very* difficult at first, but stick with it. If you go through this book and do each exercise for one or two hours a night, you will have a good foundation for moving on to another book. You might not really learn “programming” from this book, but you will learn the foundation skills you need to start learning the language.

This book’s job is to teach you the three most essential skills that a beginning programmer needs to know: reading and writing, attention to detail, and spotting differences.

Reading and Writing

It seems stupidly obvious, but if you have a problem typing, you will have a problem learning to code. Especially if you have a problem typing the fairly odd characters in source code. Without this simple skill, you will be unable to learn even the most basic things about how software works.

Typing the code samples and getting them to run will help you learn the names of the symbols, get you familiar with typing them, and get you reading the language.

Attention to Detail

The one skill that separates bad programmers from good programmers is attention to detail. In fact, it’s what separates the good from the bad in any profession. Without paying attention to the tiniest details of your work, you will miss key elements of what you create. In programming, this is how you end up with bugs and difficult-to-use systems.

By going through this book and copying each example *exactly*, you will be training your brain to focus on the details of what you are doing, as you are doing it.

Spotting Differences

A very important skill—which most programmers develop over time—is the ability to visually notice differences between things. An experienced programmer can take two pieces of code that are slightly different and immediately start pointing out the differences. Programmers have invented tools to make this even easier, but we won’t be using any of these. You first have to train your brain the hard way—then you can use the tools.

While you do these exercises, typing each one in, you will make mistakes. It’s inevitable; even seasoned programmers make a few. Your job is to compare what you have written to what’s required and fix all the differences. By doing so, you will train yourself to notice mistakes, bugs, and other problems.

Do Not Copy-Paste

You must *type* each of these exercises in, manually. If you copy and paste, you might as well just not even do them. The point of these exercises is to train your hands, your brain, and your mind

If you are reading this book and flipping out at every third sentence because you feel I'm insulting your intelligence, then I have three points of advice for you:

1. Stop reading my book. I didn't write it for you. I wrote it for people who don't already know everything.
2. Empty before you fill. You will have a hard time learning from someone with more knowledge if you already know everything.
3. Go learn Lisp. I hear people who know everything really like Lisp.

For everyone else who's here to learn, just read everything as if I'm smiling and I have a mischievous little twinkle in my eye.

EXERCISE 0

The Setup

This exercise has no code. It is simply the exercise you complete to get your computer to run Python. You should follow these instructions as exactly as possible. For example, Mac OSX computers already have Python 2, so do not install Python 3 (or any Python).

WARNING! If you do not know how to use PowerShell on Windows or the Terminal on OSX or "Bash" on Linux, then you need to go learn that first. I have included an abbreviated version of my book *The Command Line Crash Course* in the appendix. Go through that first and then come back to these instructions.

Mac OSX

To complete this exercise, complete the following tasks:

1. Go to http://www.barebones.com/products/textwrangler with your browser, get the TextWrangler text editor, and install it.
2. Put TextWrangler (your editor) in your dock so you can reach it easily.
3. Find your Terminal program. Search for it. You will find it.
4. Put your Terminal in your dock as well.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `python`. You run things in Terminal by just typing the name and hitting RETURN.
7. Hit CTRL-Z (^Z), `Enter`, and get out of `python`.
8. You should be back at a prompt similar to what you had before you typed `python`. If not, find out why.
9. Learn how to make a directory in the Terminal.
10. Learn how to change into a directory in the Terminal.
11. Use your editor to create a file in this directory. You will make the file, "Save" or "Save As . . . ," and pick this directory.
12. Go back to Terminal using just the keyboard to switch windows.
13. Back in Terminal, see if you can list the directory to see your newly created file.

OSX: What You Should See

Here's me doing this on my computer in Terminal. Your computer would be different, so see if you can figure out all the differences between what I did and what you should do.

```
Last login: Sat Apr 24 00:56:54 on ttys001
~ $ python
Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
~ $ mkdir mystuff
~ $ cd mystuff
mystuff $ ls
# ... Use TextWrangler here to edit test.txt....
mystuff $ ls
test.txt
mystuff $
```

Windows

1. Go to http://notepad-plus-plus.org with your browser, get the `Notepad++` text editor, and install it. You do not need to be the administrator to do this.
2. Make sure you can get to `Notepad++` easily by putting it on your desktop and/or in `Quick Launch`. Both options are available during setup.
3. Run PowerShell from the Start menu. Search for it and you can just hit Enter to run it.
4. Make a shortcut to it on your desktop and/or `Quick Launch` for your convenience.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `python`. You run things in Terminal by just typing the name and hitting Enter.
   a. If you run `python` and it's not there (`python is not recognized.`), install it from http://python.org/download.
   b. *Make sure you install Python 2, not Python 3.*
   c. You may be better off with ActiveState Python, especially if you do not have administrative rights.
   d. If after you install it `python` still isn't recognized, then in PowerShell enter this:

```
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27", "User")
```

   e. Close PowerShell and then start it again to make sure Python now runs. If it doesn't, restart may be required.

7. Type `quit()` and hit `Enter` to exit python.
8. You should be back at a prompt similar to what you had before you typed `python`. If not, find out why.
9. Learn how to make a directory in the Terminal.
10. Learn how to change into a directory in the Terminal.
11. Use your editor to create a file in this directory. Make the file, `Save` or `Save As...` and pick this directory.
12. Go back to Terminal using just the keyboard to switch windows.
13. Back in Terminal, see if you can list the directory to see your newly created file.

WARNING! If you missed it, sometimes you install Python on Windows and it doesn't configure the path correctly. Make sure you enter `[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27", "User")` in `PowerShell` to configure it correctly. You also have to either restart PowerShell or restart your whole computer to get it to really be fixed.

Windows: What You Should See

```
 > python
 ActivePython 2.6.5.12 (ActiveState Software Inc.) based on
 Python 2.6.5 (r265:79063, Mar 20 2010, 14:22:52) [MSC v.1500 32 bit (Intel)] on win32
 Type "help", "copyright", "credits" or "license" for more information.
 >>> ^Z


> mkdir mystuff

> cd mystuff

... Here you would use Notepad++ to make test.txt in mystuff ...

 >
  <bunch of unimportant errors if you installed it as non-admin - ignore them - hit Enter>
> dir
 Volume in drive C is
 Volume Serial Number is 085C-7E02

 Directory of C:\Documents and Settings\you\mystuff

04.05.2010  23:32    <DIR>          .
04.05.2010  23:32    <DIR>          ..
04.05.2010  23:32                 6 test.txt
```

```
               1 File(s)              6 bytes
               2 Dir(s)  14 804 623 360 bytes free

>
```

You will probably see a very different prompt, Python information, and other stuff, but this is the general idea.

Linux

Linux is a varied operating system with a bunch of different ways to install software. I'm assuming if you are running Linux then you know how to install packages, so here are your instructions:

1. Use your Linux package manager and install the gedit text editor.
2. Make sure you can get to gedit easily by putting it in your window manager's menu.
   a. Run gedit so we can fix some stupid defaults it has.
   b. Open `Preferences` and select the `Editor` tab.
   c. Change `Tab width:` to 4.
   d. Select (make sure a check mark is in) `Insert spaces instead of tabs`.
   e. Turn on `Automatic indentation` as well.
   f. Open the `View` tab and turn on `Display line numbers.`
3. Find your Terminal program. It could be called GNOME Terminal, Konsole, or xterm.
4. Put your Terminal in your dock as well.
5. Run your Terminal program. It won't look like much.
6. In your Terminal program, run `Python`. You run things in Terminal by just typing the name and hitting `Enter`.
   a. If you run `Python` and it's not there, install it. *Make sure you install Python 2, not Python 3.*
7. Type `quit()` and hit `Enter` to exit Python.
8. You should be back at a prompt similar to what you had before you typed `python`. If not, find out why.
9. Learn how to make a directory in the Terminal.
10. Learn how to change into a directory in the Terminal.
11. Use your editor to create a file in this directory. Typically you will make the file, Save or `Save As...`, and pick this directory.

12. Go back to Terminal using just the keyboard to switch windows. Look it up if you can't figure it out.
13. Back in Terminal, see if you can list the directory to see your newly created file.

Linux: What You Should See

```
$ python
Python 2.6.5 (r265:79063, Apr  1 2010, 05:28:39)
[GCC 4.4.3 20100316 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
$ mkdir mystuff
$ cd mystuff
# ... Use gedit here to edit test.txt ...
$ ls
test.txt
$
```

You will probably see a very different prompt, Python information, and other stuff, but this is the general idea.

Warnings for Beginners

You are done with this exercise. This exercise might be hard for you, depending on your familiarity with your computer. If it is difficult, take the time to read and study and get through it, because until you can do these very basic things, you will find it difficult to get much programming done.

If a programmer tells you to use vim or emacs, just say "no." These editors are for when you are a better programmer. All you need right now is an editor that lets you put text into a file. We will use gedit, TextWrangler, or Notepad++ (from now on called "the text editor" or "a text editor") because it is simple and the same on all computers. Professional programmers use these text editors, so it's good enough for you starting out.

A programmer may try to get you to install Python 3 and learn that. Say, "When all the Python code on your computer is Python 3, then I'll try to learn it." That should keep him or her busy for about 10 years.

A programmer will eventually tell you to use Mac OSX or Linux. If the programmer likes fonts and typography, he'll tell you to get a Mac OSX computer. If he likes control and has a huge beard, he'll tell you to install Linux. Again, use whatever computer you have right now that works. All you need is an editor, a Terminal, and Python.

Finally, the purpose of this setup is so you can do four things very reliably while you work on the exercises:

1. *Write* exercises using your text editor, gedit on Linux, TextWrangler on OSX, or Notepad++ on Windows.
2. *Run* the exercises you wrote.
3. *Fix* them when they are broken.
4. Repeat.

Anything else will only confuse you, so stick to the plan.

EXERCISE 1

A Good First Program

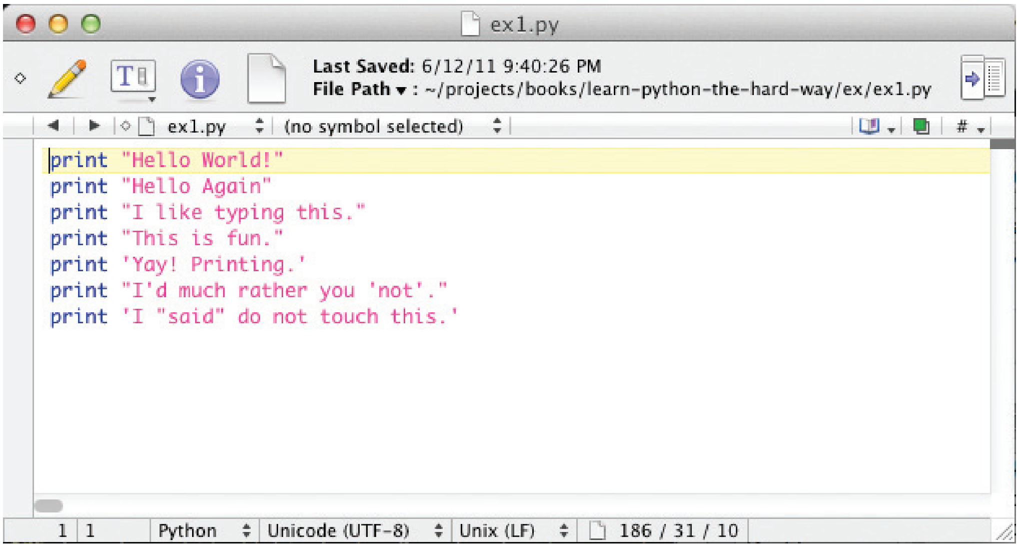
Remember, you should have spent a good amount of time in Exercise 0, learning how to install a text editor, run the text editor, run the Terminal, and work with both of them. If you haven't done that, then do not go on. You will not have a good time. This is the only time I'll start an exercise with a warning that you should not skip or get ahead of yourself.

Type the following text into a single file named `ex1.py`. This is important, as Python works best with files ending in `.py`.

ex1.py

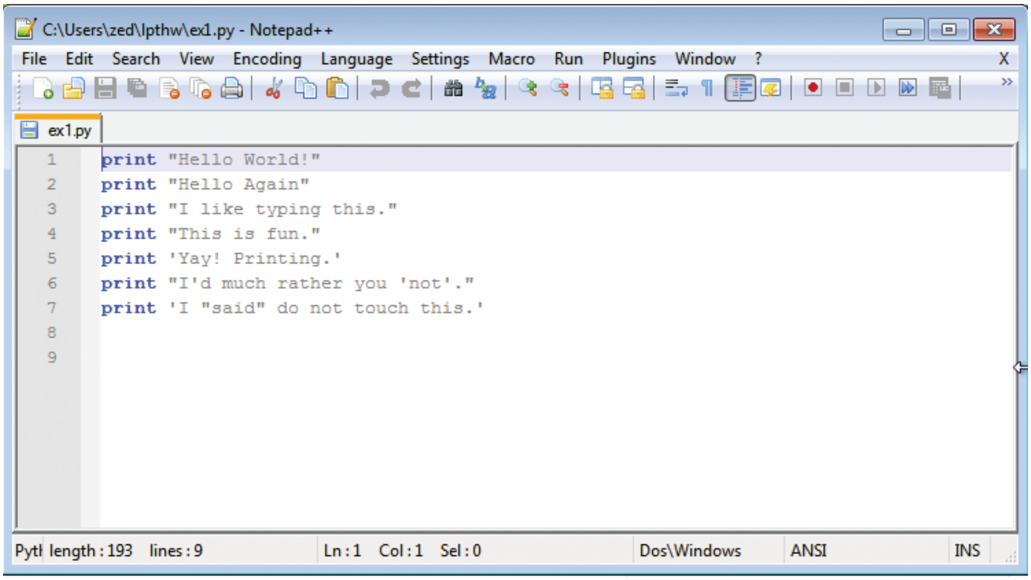
```python
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

If you are on Mac OSX, then this is what your text editor might look like if you use TextWrangler:


ex1.py
Last Saved: 6/12/11 9:40:26 PM
File Path ▾ : ~/projects/books/learn-python-the-hard-way/ex/ex1.py
ex1.py
(no symbol selected)
1 1
Python
Unicode (UTF-8)
Unix (LF)
186 / 31 / 10

```python
print "Hello World!"
print "Hello Again"
print "I like typing this."
print "This is fun."
print 'Yay! Printing.'
print "I'd much rather you 'not'."
print 'I "said" do not touch this.'
```

If you are on Windows using Notepad++, then this is what it would look like:


C:\Users\zed\lpthw\ex1.py - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
ex1.py
Pytl length : 193 lines : 9 Ln : 1 Col : 1 Sel : 0 Dos\Windows ANSI INS

```
1   print "Hello World!"
2   print "Hello Again"
3   print "I like typing this."
4   print "This is fun."
5   print 'Yay! Printing.'
6   print "I'd much rather you 'not'."
7   print 'I "said" do not touch this.'
8
9
```

Don't worry if your editor doesn't look exactly the same; the key points are as follows:

1. Notice I did not type the line numbers on the left. Those are printed in the book so I can talk about specific lines by saying, "See line 5 . . ." You do not type those into Python scripts.

2. Notice I have the `print` at the beginning of the line and how it looks exactly the same as what I have above. Exactly means exactly, not kind of sort of the same. Every single character has to match for it to work. But the colors are all different. Color doesn't matter; only the characters you type.

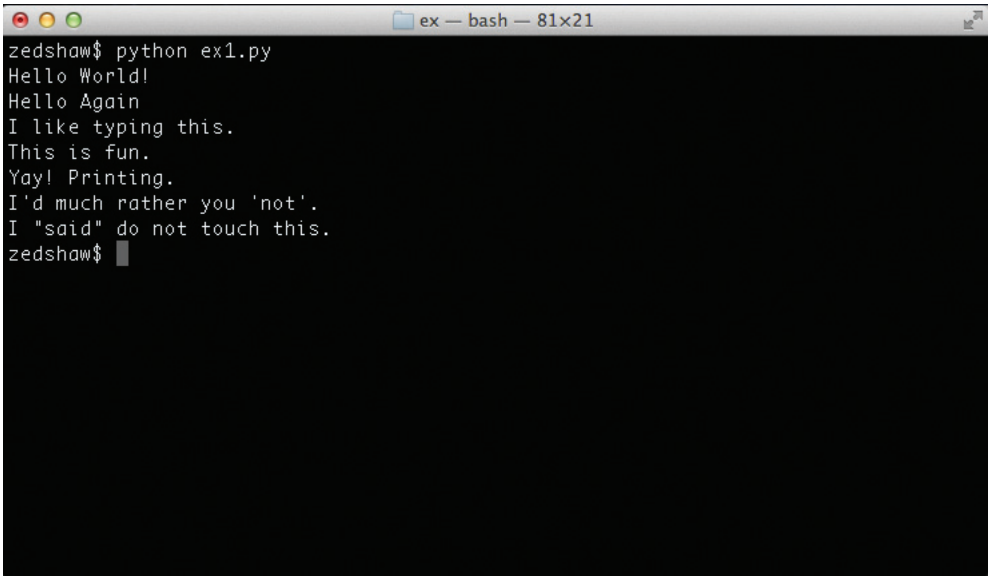
Then in Terminal, *run* the file by typing:

```
python ex1.py
```

If you did it right, then you should see the same output I have below. If not, you have done something wrong. No, the computer is not wrong.

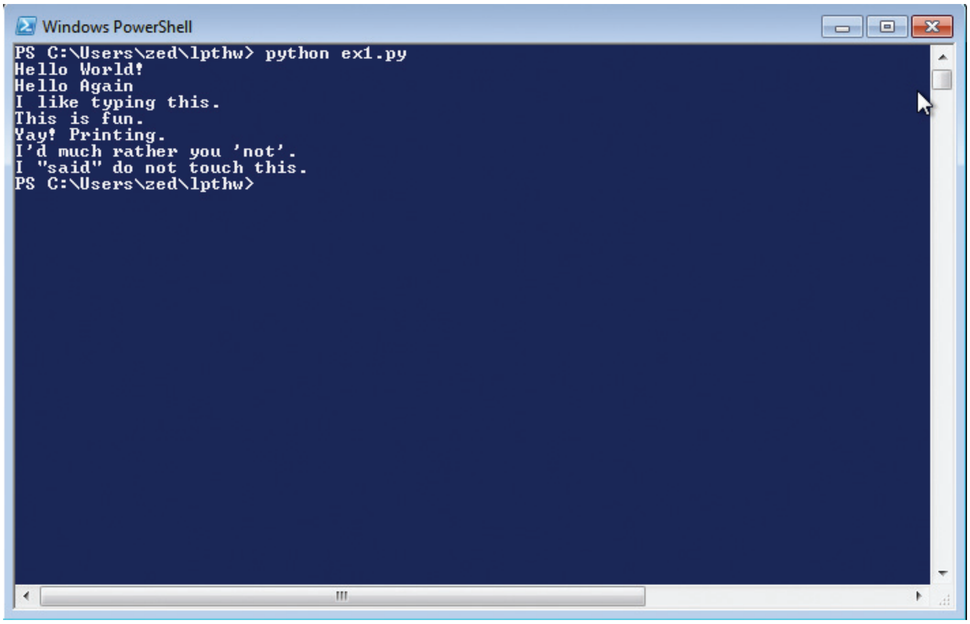
What You Should See

On Max OSX in the Terminal, you should see this:



```
zedshaw$ python ex1.py
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
zedshaw$
```

On Windows in PowerShell, you should see this:



```
PS C:\Users\zed\lpthw> python ex1.py
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
PS C:\Users\zed\lpthw>
```

You may see different names, the name of your computer or other things, before the `python ex1.py`, but the important part is that you type the command and see the output is the same as mine.

If you have an error, it will look like this:

```
$ python ex/ex1.py
  File "ex/ex1.py", line 3
    print "I like typing this.
                             ^
SyntaxError: EOL while scanning string literal
```

It's important that you can read these, since you will be making many of these mistakes. Even I make many of these mistakes. Let's look at this line by line.

1. Here we ran our command in the Terminal to run the `ex1.py` script.
2. Python then tells us that the file `ex1.py` has an error on line 3.
3. It then prints this line for us.
4. Then it puts a ^ (caret) character to point at where the problem is. Notice the missing " (double-quote) character?
5. Finally, it prints out a `SyntaxError` and tells us something about what might be the error. Usually these are very cryptic, but if you copy that text into a search engine, you will find someone else who's had that error and you can probably figure out how to fix it.

WARNING! If you are from another country and you get errors about ASCII encodings, then put this at the top of your Python scripts:

```
# -*- coding: utf-8 -*-
```

It will fix them so that you can use Unicode UTF-8 in your scripts without a problem.

Study Drills

Each exercise also contains Study Drills. The Study Drills contain things you should *try* to do. If you can't, skip it and come back later.

For this exercise, try these things:

1. Make your script print another line.
2. Make your script print only one of the lines.
3. Put a "#" (octothorpe) character at the beginning of a line. What did it do? Try to find out what this character does.

From now on, I won't explain how each exercise works unless an exercise is different.

NOTE: An "octothorpe" is also called a "pound," "hash," "mesh," or any number of names. Pick the one that makes you chill out.

Common Student Questions

These are *actual* questions by real students in the comments section of the book when it was online. You may run into some of these, so I've collected and answered them for you.

Can I use IDLE?
No, you should use Terminal on OSX and PowerShell on Windows, just like I have here. If you don't know how to use those, then you can go read the Command Line Crash Course in the appendix.

How do you get colors in your editor?
Save your file first as a `.py` file, such as `ex1.py`. Then you'll have color when you type.

I get `SyntaxError: invalid syntax` when I run ex1.py.
You are probably trying to run Python, then trying to type Python again. Close your Terminal, start it again, and right away type only `python ex1.py`.

I get `can't open file 'ex1.py': [Errno 2] No such file or directory`.
You need to be in the same directory as the file you created. Make sure you use the `cd` command to go there first. For example, if you saved your file in `lpthw/ex1.py`, then you would do `cd lpthw/` before trying to run `python ex1.py`. If you don't know what any of that means, then go through the Command Line Crash Course (CLI-CC) mentioned in the first question.

How do I get my country's language characters into my file?
Make sure you type this at the top of your file: `# -*- coding: utf-8 -*-`.

My file doesn't run; I just get the prompt back with no output.
You most likely took the previous code literally and thought that `print "Hello World!"` meant to literally print just `"Hello World!"` into the file, without the `print`. Your file has to be *exactly* like mine in the previous code and all the screenshots; I have `print "Hello World!"` and `print` before every line. Make sure your code is like mine and it should work.

EXERCISE 2

Comments and Pound Characters

Comments are very important in your programs. They are used to tell you what something does in English, and they also are used to disable parts of your program if you need to remove them temporarily. Here's how you use comments in Python:

ex2.py

```
# A comment, this is so you can read your program later.
# Anything after the # is ignored by python.

print "I could have code like this." # and the comment after is ignored

# You can also use a comment to "disable" or comment out a piece of code:
# print "This won't run."

print "This will run."
```

From now on, I'm going to write code like this. It is important for you to understand that everything does not have to be literal. Your screen and program may visually look different, but what's important is the text you type into the file you're writing in your text editor. In fact, I could work with any text editor and the results would be the same.

What You Should See

Exercise 2 Session

```
$ python ex2.py
I could have code like this.
This will run.
```

Again, I'm not going to show you screenshots of all the Terminals possible. You should understand that the above is not a literal translation of what your output should look like visually, but the text between the first `$ Python ...` and last $ lines will be what you focus on.

Study Drills

1. Find out if you were right about what the # character does and make sure you know what it's called (octothorpe or pound character).
2. Take your `ex2.py` file and review each line going backward. Start at the last line, and check each word in reverse against what you should have typed.

3. Did you find more mistakes? Fix them.
4. Read what you typed previously out loud, including saying each character by its name. Did you find more mistakes? Fix them.

Common Student Questions

Are you sure # is called the pound character?
I call it the octothorpe and that is the only name that no country uses and that works in every country. Every country thinks its way to call this one character is both the most important way to do it and also the only way it's done. To me this is simply arrogance and, really, y'all should just chill out and focus on more important things like learning to code.

If # is for comments, then how come `# -*- coding: utf-8 -*-` works?
Python still ignores that as code, but it's used as a kind of "hack" or workaround for problems with setting and detecting the format of a file. You also find a similar kind of comment for editor settings.

Why does the # in `print "Hi # there."` not get ignored?
The # in that code is inside a string, so it will be put into the string until the ending " character is hit. These pound characters are just considered characters and aren't considered comments.

How do I comment out multiple lines?
Put a # in front of each one.

I can't figure out how to type a # character on my country's keyboard?
Some countries use the Alt key and combinations of those to print characters foreign to their language. You'll have to look online in a search engine to see how to type it.

Why do I have to read code backward?
It's a trick to make your brain not attach meaning to each part of the code, and doing that makes you process each piece exactly. This catches errors and is a handy error-checking technique.

EXERCISE 3

Numbers and Math

Every programming language has some kind of way of doing numbers and math. Do not worry: programmers lie frequently about being math geniuses when they really aren't. If they were math geniuses, they would be doing math, not writing ads and social network games to steal people's money.

This exercise has lots of math symbols. Let's name them right away so you know what they are called. As you type this one in, say the names. When saying them feels boring, you can stop saying them. Here are the names:

+	plus
-	minus
/	slash
*	asterisk
%	percent
<	less-than
>	greater-than
<=	less-than-equal
>=	greater-than-equal

Notice how the operations are missing? After you type in the code for this exercise, go back and figure out what each of these does and complete the table. For example, + does addition.

ex3.py

```python
print "I will now count my chickens:"

print "Hens", 25 + 30 / 6
print "Roosters", 100 - 25 * 3 % 4

print "Now I will count the eggs:"

print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6

print "Is it true that 3 + 2 < 5 - 7?"

print 3 + 2 < 5 - 7

print "What is 3 + 2?", 3 + 2
print "What is 5 - 7?", 5 - 7

print "Oh, that's why it's False."

print "How about some more."

print "Is it greater?", 5 > -2
print "Is it greater or equal?", 5 >= -2
print "Is it less or equal?", 5 <= -2
```

What You Should See

Exercise 3 Session

```
$ python ex3.py
I will now count my chickens:
Hens 30
Roosters 97
Now I will count the eggs:
7
Is it true that 3 + 2 < 5 - 7?
False
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's False.
How about some more.
Is it greater? True
Is it greater or equal? True
Is it less or equal? False
```

Study Drills

1. Above each line, use the # to write a comment to yourself explaining what the line does.
2. Remember in Exercise 0 when you started Python? Start Python this way again and, using the above characters and what you know, use Python as a calculator.
3. Find something you need to calculate and write a new `.py` file that does it.
4. Notice the math seems “wrong”? There are no fractions, only whole numbers. Find out why by researching what a “floating point” number is.
5. Rewrite `ex3.py` to use floating point numbers so it’s more accurate (hint: 20.0 is floating point).

Common Student Questions

Why is the % character a "modulus" and not a "percent"?
Mostly that's just how the designers chose to use that symbol. In normal writing, you are correct to read it as a "percent." In programming, this calculation is typically done with simple division and the / operator. The % modulus is a different operation that just happens to use the % symbol.

How does % work?
Another way to say it is "X divided by Y with J remaining." For example, "100 divided by 16 with 4 remaining." The result of % is the J part, or the remaining part.

What is the order of operations?
In the United States we use an acronym called PEMDAS, which stands for Parentheses Exponents Multiplication Division Addition Subtraction. That's the order Python follows as well.

Why does / (divide) round down?
It's not really rounding down; it's just dropping the fractional part after the decimal. Try doing `7.0 / 4.0` and compare it to `7 / 4` and you'll see the difference.

This page intentionally left blank

Variables and Names

Now you can print things with `print` and you can do math. The next step is to learn about variables. In programming, a variable is nothing more than a name for something so you can use the name rather than the something as you code. Programmers use these variable names to make their code read more like English and because they have lousy memories. If they didn't use good names for things in their software, they'd get lost when they tried to read their code again.

If you get stuck with this exercise, remember the tricks you have been taught so far of finding differences and focusing on details:

1. Write a comment above each line explaining to yourself what it does in English.
2. Read your `.py` file backward.
3. Read your `.py` file out loud, saying even the characters.

ex4.py

```python
cars = 100
space_in_a_car = 4.0
drivers = 30
passengers = 90
cars_not_driven = cars - drivers
cars_driven = drivers
carpool_capacity = cars_driven * space_in_a_car
average_passengers_per_car = passengers / cars_driven


print "There are", cars, "cars available."
print "There are only", drivers, "drivers available."
print "There will be", cars_not_driven, "empty cars today."
print "We can transport", carpool_capacity, "people today."
print "We have", passengers, "to carpool today."
print "We need to put about", average_passengers_per_car, "in each car."
```

NOTE: The _ in `space_in_a_car` is called an underscore character. Find out how to type it if you do not already know. We use this character a lot to put an imaginary space between words in variable names.

What You Should See

Exercise 4 Session

```
$ python ex4.py
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3 in each car.
```

Study Drills

When I wrote this program the first time I had a mistake, and Python told me about it like this:

```
Traceback (most recent call last):
       File "ex4.py", line 8, in <module>
         average_passengers_per_car = car_pool_capacity / passenger
     NameError: name 'car_pool_capacity' is not defined
```

Explain this error in your own words. Make sure you use line numbers and explain why.

Here's more Study Drills:

1. I used 4.0 for `space_in_a_car`, but is that necessary? What happens if it's just 4?
2. Remember that 4.0 is a "floating point" number. Find out what that means.
3. Write comments above each of the variable assignments.
4. Make sure you know what = is called (equals) and that it's making names for things.
5. Remember that _ is an underscore character.
6. Try running Python as a calculator like you did before and use variable names to do your calculations. Popular variable names are also `i`, `x`, and `j`.

Common Student Questions

What is the difference between = (single-equal) and == (double-equal)?
The = (single-equal) assigns the value on the right to a variable on the left. The == (double-equal) tests if two things have the same value, and you'll learn about this in Exercise 27.

Can we write x=100 instead of `x = 100`?
You can, but it's bad form. You should add space around operators like this so that it's easier to read.

How can I print without spaces between words in `print`?
You do it like this: `print "Hey %s there." % "you"`. You will do more of this soon.

What do you mean by "read the file backward"?
Very simple. Imagine you have a file with 16 lines of code in it. Start at line 16, and compare it to my file at line 16. Then do it again for 15, and so on, until you've read the whole file backward.

Why did you use `4.0` for space?
It is mostly so you can then find out what a floating point number is and ask this question. See the Study Drills.

EXERCISE 5

More Variables and Printing

Now we'll do even more typing of variables and printing them out. This time we'll use something called a "format string." Every time you put " (double-quotes) around a piece of text, you have been making a *string*. A string is how you make something that your program might give to a human. You print them, save them to files, send them to web servers, all sorts of things.

Strings are really handy, so in this exercise you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using specialized format sequences and then putting the variables at the end with a special syntax that tells Python, "Hey, this is a format string, put these variables in there."

As usual, just type this in even if you do not understand it and make it exactly the same.

ex5.py

```python
my_name = 'Zed A. Shaw'
my_age = 35 # not a lie
my_height = 74 # inches
my_weight = 180 # lbs
my_eyes = 'Blue'
my_teeth = 'White'
my_hair = 'Brown'

print "Let's talk about %s." % my_name
print "He's %d inches tall." % my_height
print "He's %d pounds heavy." % my_weight
print "Actually that's not too heavy."
print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
print "His teeth are usually %s depending on the coffee." % my_teeth

# this line is tricky, try to get it exactly right
print "If I add %d, %d, and %d I get %d." % (
    my_age, my_height, my_weight, my_age + my_height + my_weight)
```

WARNING! Remember to put `# -- coding: utf-8 --` at the top if you use non-ASCII characters and get an encoding error.

What You Should See

Exercise 5 Session

```
$ python ex5.py
Let's talk about Zed A. Shaw.
He's 74 inches tall.
```

```
He's 180 pounds heavy.
Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.
```

Study Drills

1. Change all the variables so there isn't the `my_` in front. Make sure you change the name everywhere, not just where you used = to set them.
2. Try more format characters. `%r` is a very useful one. It's like saying "print this no matter what."
3. Search online for all the Python format characters.
4. Try to write some variables that convert the inches and pounds to centimeters and kilos. Do not just type in the measurements. Work out the math in Python.

Common Student Questions

Can I make a variable like this: `1 = 'Zed Shaw'`?
No, the 1 is not a valid variable name. They need to start with a character, so a1 would work, but 1 will not.

What does %s, %r, and %d do again?
You'll learn more about this as you continue, but they are "formatters." They tell Python to take the variable on the right and put it in to replace the %s with its value.

I don't get it, what is a "formatter"? Huh?
The problem with teaching you programming is that to understand many of my descriptions, you need to know how to do programming already. The way I solve this is I make you do something, and then I explain it later. When you run into these kinds of questions, write them down and see if I explain it later.

How can I round a floating point number?
You can use the **`round()`** function like this: **`round(1.7333)`**.

I get this error TypeError: `'str' object is not callable`.
You probably forgot the % between the string and the list of variables.

Why does this not make sense to me?
Try making the numbers in this script your measurements. It's weird, but talking about yourself will make it seem more real.

EXERCISE 6

Strings and Text

While you have already been writing strings, you still do not know what they do. In this exercise, we create a bunch of variables with complex strings so you can see what they are for. First an explanation of strings.

A string is usually a bit of text you want to display to someone or "export" out of the program you are writing. Python knows you want something to be a string when you put either `"` (double-quotes) or `'` (single-quotes) around the text. You saw this many times with your use of `print` when you put the text you want to go to the string inside `"` or `'` after the `print`. Then Python prints it.

Strings may contain the format characters you have discovered so far. You simply put the formatted variables in the string, and then a % (percent) character, followed by the variable. The *only* catch is that if you want multiple formats in your string to print multiple variables, you need to put them inside `( )` (parentheses) separated by `,` (commas). It's as if you were telling me to buy you a list of items from the store and you said, "I want milk, eggs, bread, and soup." Only as a programmer we say, "(milk, eggs, bread, soup)."

We will now type in a whole bunch of strings, variables, and formats, and print them. You will also practice using short abbreviated variable names. Programmers love saving themselves time at your expense by using annoying cryptic variable names, so let's get you started being able to read and write them early on.

ex6.py

```
x = "There are %d types of people." % 10
binary = "binary"
do_not = "don't"
y = "Those who know %s and those who %s." % (binary, do_not)

print x
print y

print "I said: %r." % x
print "I also said: '%s'." % y

hilarious = False
joke_evaluation = "Isn't that joke so funny?! %r"

print joke_evaluation % hilarious

w = "This is the left side of..."
e = "a string with a right side."

print w + e
```

What You Should See

Exercise 6 Session

```
$ python ex6.py
There are 10 types of people.
Those who know binary and those who don't.
I said: 'There are 10 types of people.'.
I also said: 'Those who know binary and those who don't.'.
Isn't that joke so funny?! False
This is the left side of...a string with a right side.
```

Study Drills

1. Go through this program and write a comment above each line explaining it.
2. Find all the places where a string is put inside a string. There are four places.
3. Are you sure there are only four places? How do you know? Maybe I like lying.
4. Explain why adding the two strings `w` and `e` with `+` makes a longer string.

Common Student Questions

What is the difference between `%r` and `%s`?
We use `%r` for debugging, since it displays the “raw” data of the variable, but we use `%s` and others for displaying to users.

What’s the point of `%s` and `%d` when you can just use `%r`?
The `%r` is best for debugging, and the other formats are for actually displaying variables to users.

If you thought the joke was funny could you write `hilarious = True`?
Yes, and you’ll learn more about these boolean values in Exercise 27.

Why do you put `'` (single-quotes) around some strings and not others?
Mostly it’s because of style, but I’ll use a single-quote inside a string that has double-quotes. Look at line 10 to see how I’m doing that.

I get the error `TypeError: not all arguments converted during string formatting`.
You need to make sure that the line of code is exactly the same. What happens in this error is you have more `%` format characters in the string than variables to put in them. Go back and figure out what you did wrong.

EXERCISE 7

More Printing

Now we are going to do a bunch of exercises where you just type code in and make it run. I won't be explaining much since it is just more of the same. The purpose is to build up your chops. See you in a few exercises, and *do not skip!* Do not *paste!*

ex7.py

```python
print "Mary had a little lamb."
print "Its fleece was white as %s." % 'snow'
print "And everywhere that Mary went."
print "." * 10  # what'd that do?

end1 = "C"
end2 = "h"
end3 = "e"
end4 = "e"
end5 = "s"
end6 = "e"
end7 = "B"
end8 = "u"
end9 = "r"
end10 = "g"
end11 = "e"
end12 = "r"

# watch that comma at the end.  try removing it to see what happens
print end1 + end2 + end3 + end4 + end5 + end6,
print end7 + end8 + end9 + end10 + end11 + end12
```

What You Should See

Exercise 7 Session

```
$ python ex7.py
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.
..........
Cheese Burger
```

Study Drills

For these next few exercises, you will have the exact same Study Drills.

1. Go back through and write a comment on what each line does.
2. Read each one backward or out loud to find your errors.
3. From now on, when you make mistakes, write down on a piece of paper what kind of mistake you made.
4. When you go to the next exercise, look at the last mistakes you made and try not to make them in this new one.
5. Remember that everyone makes mistakes. Programmers are like magicians who like everyone to think they are perfect and never wrong, but it's all an act. They make mistakes all the time.

Common Student Questions

How does the "end" statement work?
These are not really an "end statement," but actually the names of variables that just happen to have the word "end" in them.

Why are you using the variable named `'snow'`?
That's actually not a variable: it is just a string with the word `snow` in it. A variable wouldn't have the single-quotes around it.

Is it normal to write an English comment for every line of code like you say to do in Study Drills #1?
No, normally you write comments only to explain difficult to understand code or why you did something. Why (or your motivation) is usually much more important, and then you try to write the code so that it explains how something is being done on its own. However, sometimes you just have to write such nasty code to solve a problem that it does need a comment on every line. In this case, though, it's strictly for you to get better at translating from code to English.

Can I use single-quotes or double-quotes to make a string or do they do different things?
In Python either way to make a string is acceptable, although typically you'll use single-quotes for any short strings like `'a'` or `'snow'`.

Couldn't you just not use the comma `,` and turn the last two lines into one single-line print?
Yes, you could very easily, but then it'd be longer than 80 characters, which in Python is bad style.

EXERCISE 8

Printing, Printing

ex8.py

```python
formatter = "%r %r %r %r"

print formatter % (1, 2, 3, 4)
print formatter % ("one", "two", "three", "four")
print formatter % (True, False, False, True)
print formatter % (formatter, formatter, formatter, formatter)
print formatter % (
    "I had this thing.",
    "That you could type up right.",
    "But it didn't sing.",
    "So I said goodnight."
)
```

What You Should See

Exercise 8 Session

```
$ python ex8.py
1 2 3 4
'one' 'two' 'three' 'four'
True False False True
'%r %r %r %r' '%r %r %r %r' '%r %r %r %r' '%r %r %r %r'
'I had this thing.' 'That you could type up right.' "But it didn't sing."
'So I said goodnight.'
```

Study Drills

1. Do your checks of your work, write down your mistakes, and try not to make them on the next exercise.
2. Notice that the last line of output uses both single-quotes and double-quotes for individual pieces. Why do you think that is?

Common Student Questions

Should I use %s or %r for formatting?
You should use %s and only use %r for getting debugging information about something. The %r will give you the "raw programmer's" version of variable, also known as the "representation."

Why do I have to put quotes around "one" but not around `True` or `False`?
That's because Python recognizes `True` and `False` as keywords representing the concept of true and false. If you put quotes around them, then they are turned into strings and won't work right. You'll learn more about how these work in Exercise 27.

I tried putting Chinese (or some other non-ASCII characters) into these strings, but `%r` prints out weird symbols.
Use `%s` to print that instead and it'll work.

Why does `%r` sometimes print things with single-quotes when I wrote them with double-quotes?
Python is going to print the strings in the most efficient way it can, not replicate exactly the way you wrote them. This is perfectly fine since `%r` is used for debugging and inspection, so it's not necessary that it be pretty.

Why doesn't this work in Python 3?
Don't use Python 3. Use Python 2.7 or better, although Python 2.6 might work fine.

Can I use IDLE to run this?
No, you should learn to use the command line. It is essential to learning programming and is a good place to start if you want to learn about programming. IDLE will fail for you when you get further in the book.

EXERCISE 9

Printing, Printing, Printing

ex9.py

```
# Here's some new strange stuff, remember type it exactly.

days = "Mon Tue Wed Thu Fri Sat Sun"
months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"

print "Here are the days: ", days
print "Here are the months: ", months

print """
There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
"""
```

What You Should See

Exercise 9 Session

```
$ python ex9.py
Here are the days:  Mon Tue Wed Thu Fri Sat Sun
Here are the months:  Jan
Feb
Mar
Apr
May
Jun
Jul
Aug

There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

Study Drills

1. Do your checks of your work, write down your mistakes, and try not to make them on the next exercise.

Common Student Questions

What if I wanted to start the months on a new line?
You simply start the string with `\n` like this:

```
"\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
```

Why do the `\n` newlines not work when I use `%r`?
That's how `%r` formatting works; it prints it the way you wrote it (or close to it). It's the "raw" format for debugging.

Why do I get an error when I put spaces between the three double-quotes?
You have to type them like `"""` and not `"  "  "`, meaning with *no* spaces between each one.

Is it bad that my errors are always spelling mistakes?
Most programming errors in the beginning (and even later) are simple spelling mistakes, typos, or getting simple things out of order.

What Was That?

In Exercise 9 I threw you some new stuff, just to keep you on your toes. I showed you two ways to make a string that goes across multiple lines. In the first way, I put the characters `\n` (backslash n) between the names of the months. What these two characters do is put a `new line character` into the string at that point.

This use of the `\` (backslash) character is a way we can put difficult-to-type characters into a string. There are plenty of these "escape sequences" available for different characters you might want to put in, but there's a special one, the `double backslash`, which is just two of them `\`. These two characters will print just one backslash. We'll try a few of these sequences so you can see what I mean.

Another important escape sequence is to escape a single-quote `'` or double-quote `"`. Imagine you have a string that uses double-quotes and you want to put a double-quote in for the output. If you do this `"I "understand" joe."` then Python will get confused since it will think the `"` around `"understand"` actually *ends* the string. You need a way to tell Python that the `"` inside the string isn't a *real* double-quote.

To solve this problem, you *escape* double-quotes and single-quotes so Python knows what to include in the string. Here's an example:

```
"I am 6'2\" tall."  # escape double-quote inside string
'I am 6\'2" tall.'  # escape single-quote inside string
```

The second way is by using triple-quotes, which is just `"""` and works like a string, but you also can put as many lines of text as you want until you type `"""` again. We'll also play with these.

ex10.py

```
tabby_cat = "\tI'm tabbed in."
persian_cat = "I'm split\non a line."
backslash_cat = "I'm \\ a \\ cat."

fat_cat = """
I'll do a list:
\t* Cat food
\t* Fishies
\t* Catnip\n\t* Grass
"""

print tabby_cat
print persian_cat
print backslash_cat
print fat_cat
```

What You Should See

Look for the tab characters that you made. In this exercise, the spacing is important to get right.

Exercise 10 Session

```
$ python ex10.py
        I'm tabbed in.
I'm split
on a line.
I'm \ a \ cat.

I'll do a list:
        * Cat food
        * Fishies
        * Catnip
        * Grass
```

Escape Sequences

This is the list of all the escape sequences Python supports. You may not use many of these, but memorize their format and what they do anyway. Also try them out in some strings to see if you can make them work.

Escape	What it does.
`\\`	Backslash (\)
`\'`	Single-quote (')
`\"`	Double-quote (")
`\a`	ASCII bell (BEL)
`\b`	ASCII backspace (BS)
`\f`	ASCII formfeed (FF)
`\n`	ASCII linefeed (LF)
`\N{name}`	Character named name in the Unicode database (Unicode only)
`\r`	ASCII carriage return (CR)
`\t`	ASCII horizontal tab (TAB)
`\uxxxx`	Character with 16-bit hex value xxxx (Unicode only)
`\Uxxxxxxxx`	Character with 32-bit hex value xxxxxxxx (Unicode only)
`\v`	ASCII vertical tab (VT)
`\ooo`	Character with octal value oo
`\xhh`	Character with hex value hh

Here's a tiny piece of fun code to try out:

```
while True:
    for i in ["/","-","|","\\","|"]:
        print "%s\r" % i,
```

Study Drills

1. Memorize all the escape sequences by putting them on flash cards.
2. Use `'''` (triple-single-quote) instead. Can you see why you might use that instead of `"""`?
3. Combine escape sequences and format strings to create a more complex format.
4. Remember the `%r` format? Combine `%r` with double-quote and single-quote escapes and print them out. Compare %r with %s. Notice how %r prints it the way you'd write it in your file, but %s prints it the way you'd like to see it?

Common Student Questions

I still haven't completely figured out the last exercise. Should I continue?
Yes, keep going, and instead of stopping, take notes listing things you don't understand for each exercise. Periodically go through your notes and see if you can figure these things out after you've completed more exercises. Sometimes, though, you may need to go back a few exercises and go through them again.

What makes \\ special compared to the other ones?
It's simply the way you would write out one backslash (\) character. Think about why you would need this.

When I write // or /n it doesn't work.
That's because you are using a forward-slash / and not a backslash \. They are different characters that do very different things.

When I use a %r format none of the escape sequences work.
That's because `%r` is printing out the raw representation of what you typed, which is going to include the original escape sequences. Use `%s` instead. Always remember this: `%r` is for debugging; `%s` is for displaying.

I don't get Study Drills #3. What do you mean by "combine" escapes and formats?
One of the things I try to get you to understand is that each of these exercises can be combined to solve problems. Take what you know about format sequences and write some new code that uses those *and* the escapes from this exercise.

What's better, `'''` or `"""`?
It's entirely based on style. Go with the `'''` (triple-single-quote) style for now, but be ready to use either, depending on what feels best or what everyone else is doing.

Asking Questions

Now it is time to pick up the pace. I have got you doing a lot of printing so that you get used to typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have to learn to do two things that may not make sense right away, but trust me and do it anyway. It will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far you have only been printing, but you haven't been able to get any input from a person or change it. You may not even know what "input" means, so rather than talk about it, let's have you do some and see if you get it. In the next exercise, we'll do more to explain it.

ex11.py

```
print "How old are you?",
age = raw_input()
print "How tall are you?",
height = raw_input()
print "How much do you weigh?",
weight = raw_input()

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

NOTE: Notice that we put a , (comma) at the end of each `print` line. This is so that `print` doesn't end the line with a new line character and go to the next line.

What You Should See

Exercise 11 Session

```
$ python ex11.py
How old are you? 38
How tall are you? 6'2"
How much do you weigh? 180lbs
So, you're '38' old, '6\'2"' tall and '180lbs' heavy.
```

Study Drills

1. Go online and find out what Python's `raw_input` does.
2. Can you find other ways to use it? Try some of the samples you find.
3. Write another "form" like this to ask some other questions.
4. Related to escape sequences, try to find out why the last line has `'6\'2"'` with that `\'` sequence. See how the single-quote needs to be escaped because otherwise it would end the string?

Common Student Questions

How do I get a number from someone so I can do math?
That's a little advanced, but try `x = int(raw_input())`, which gets the number as a string from `raw_input()` then converts it to an integer using `int()`.

I put my height into raw input like `raw_input("6'2")` but it doesn't work.
You don't put your height in there; you type it directly into your Terminal. First thing is, go back and make the code exactly like mine. Next, run the script, and when it pauses, type your height in at your keyboard. That's all there is to it.

Why do you have a new line on line 8 instead of putting it on one line?
That's so that the line is less than 80 characters long, which is a style that Python programmers like. You could put it on one line if you like.

What's the difference between `input()` and `raw_input()`?
The `input()` function will try to convert things you enter as if they were Python code, but it has security problems so you should avoid it.

When my strings print out there's a u in front of them, as in `u'35'`.
That's how Python tells you that the string is Unicode. Use a `%s` format instead and you'll see it printed like normal.

EXERCISE 12

Prompting People

When you typed `raw_input()`, you were typing the `(` and `)` characters, which are `parenthesis` characters. This is similar to when you used them to do a format with extra variables, as in `"%s %s" % (x, y)`. For `raw_input`, you can also put in a prompt to show to a person so he knows what to type. Put a string that you want for the prompt inside the `()` so that it looks like this:

```
y = raw_input("Name? ")
```

This prompts the user with "Name?" and puts the result into the variable `y`. This is how you ask someone a question and get the answer.

This means we can completely rewrite our previous exercise using just `raw_input` to do all the prompting.

ex12.py

```
age = raw_input("How old are you? ")
height = raw_input("How tall are you? ")
weight = raw_input("How much do you weigh? ")

print "So, you're %r old, %r tall and %r heavy." % (
    age, height, weight)
```

What You Should See

Exercise 12 Session

```
$ python ex12.py
How old are you?  38
How tall are you?  6'2"
How much do you weigh?  180lbs
So, you're '38' old, '6\'2"' tall and '180lbs' heavy.
```

Study Drills

1. In Terminal, where you normally run `python` to run your scripts, type `pydoc raw_input`. Read what it says. If you're on Windows try `python -m pydoc raw_input` instead.
2. Get out of `pydoc` by typing `q` to quit.

3. Look online for what the pydoc command does.
4. Use pydoc to also read about open, `file`, `os`, and `sys`. It's alright if you do not understand those; just read through and take notes about interesting things.

Common Student Questions

How come I get `SyntaxError: invalid syntax` whenever I run pydoc?
You aren't running pydoc from the command line; you're probably running it from inside `python`. Exit out of `python` first.

Why does my pydoc not pause like yours does?
Sometimes if the help document is short enough to fit on one screen, then `pydoc` will just print it.

When I run pydoc I get `more is not recognized` as an `internal`.
Some versions of Windows do not have that command, which means pydoc is broken for you. You can skip this Study Drill and just search online for Python documentation when you need it.

Why would I use `%r` over `%s`?
Remember, `%r` is for debugging and is "raw representation" while `%s` is for display. I will not answer this question again, so you *must* memorize this fact. This is the #1 thing people ask repeatedly, and asking the same question over and over means you aren't taking the time to memorize what you should. Stop now, and finally memorize this fact.

Why can't I do `print "How old are you?" , raw_input()`?
You'd think that'd work, but Python doesn't recognize that as valid. The only answer I can really give is, you just can't.

EXERCISE 13

Parameters, Unpacking, Variables

In this exercise, we will cover one more input method you can use to pass variables to a script (script being another name for your `.py` files). You know how you type `python ex13.py` to run the `ex13.py` file? Well the `ex13.py` part of the command is called an "argument." What we'll do now is write a script that also accepts arguments.

Type this program and I'll explain it in detail:

ex13.py

```
1    from sys import argv
2
3    script, first, second, third = argv
4
5    print "The script is called:", script
6    print "Your first variable is:", first
7    print "Your second variable is:", second
8    print "Your third variable is:", third
```

On line 1 we have what's called an "import." This is how you add features to your script from the Python feature set. Rather than give you all the features at once, Python asks you to say what you plan to use. This keeps your programs small, but it also acts as documentation for other programmers who read your code later.

The `argv` is the "argument variable," a very standard name in programming that you will find used in many other languages. This variable *holds* the arguments you pass to your Python script when you run it. In the exercises you will get to play with this more and see what happens.

Line 3 "unpacks" `argv` so that, rather than holding all the arguments, it gets assigned to four variables you can work with: `script`, `first`, `second`, and `third`. This may look strange, but "unpack" is probably the best word to describe what it does. It just says, "Take whatever is in `argv`, unpack it, and assign it to all these variables on the left in order."

After that, we just print them out like normal.

Hold Up! Features Have Another Name

I call them "features" here (these little things you `import` to make your Python program do more) but nobody else calls them features. I just used that name because I needed to trick you into learning what they are without jargon. Before you can continue, you need to learn their real name: *modules*.

From now on we will be calling these "features" that we `import` *modules*. I'll say things like, "You want to import the `sys` module." They are also called "libraries" by other programmers, but let's just stick with modules.

What You Should See

Run the program like this (and you *must* pass *three* command line arguments):

Exercise 13 Session

```
$ python ex13.py first 2nd 3rd
The script is called: ex13.py
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd
```

This is what you should see when you do a few different runs with different arguments:

Exercise 13 Session

```
$ python ex13.py stuff things that
The script is called: ex13.py
Your first variable is: stuff
Your second variable is: things
Your third variable is: that
$
$ python ex13.py apple orange grapefruit
The script is called: ex13.py
Your first variable is: apple
Your second variable is: orange
Your third variable is: grapefruit
```

You can actually replace `first`, `second`, and `third` with any three things you want.

If you do not run it correctly, then you will get an error like this:

Exercise 13 Session

```
$ python ex13.py first 2nd
Traceback (most recent call last):
  File "ex13.py", line 3, in <module>
    script, first, second, third = argv
ValueError: need more than 3 values to unpack
```

This happens when you do not put enough arguments on the command when you run it (in this case just `first 2nd`). Notice when I run it I give it `first 2nd`, which caused it to give an error about "need more than 3 values to unpack," telling you that you didn't give it enough parameters.

Study Drills

1. Try giving fewer than three arguments to your script. See that error you get? See if you can explain it.
2. Write a script that has fewer arguments and one that has more. Make sure you give the unpacked variables good names.
3. Combine `raw_input` with `argv` to make a script that gets more input from a user.
4. Remember that modules give you features. Modules. Modules. Remember this because we'll need it later.

Common Student Questions

When I run it I get `ValueError: need more than 1 value to unpack`.
Remember that an important skill is paying attention to details. If you look at the What You Should See (WYSS) section, you see that I run the script with parameters on the command line. You should replicate how I ran it exactly.

What's the difference between `argv` and `raw_input()`?
The difference has to do with where the user is required to give input. If they give your script inputs on the command line, then you use `argv`. If you want them to input using the keyboard while the script is running, then use `raw_input()`.

Are the command line arguments strings?
Yes, they come in as strings, even if you typed numbers on the command line. Use `int()` to convert them just like with `raw_input()`.

How do you use the command line?
You should have learned to use it real quick by now, but if you need to learn it at this stage, then read the Command Line Crash Course appendix.

I can't combine `argv` with `raw_input()`.
Don't over think it. Just slap two lines at the end of this script that uses `raw_input()` to get something and then print it. From that, start playing with more ways to use both in the same script.

Why can't I do this `raw_input('? ') = x`?
Because that's backward. Do it the way I do it and it'll work.

Prompting and Passing

Let's do one exercise that uses argv and raw_input together to ask the user something specific. You will need this for the next exercise, where we learn to read and write files. In this exercise, we'll use raw_input slightly differently by having it just print a simple > prompt. This is similar to a game like Zork or Adventure.

ex14.py

```python
from sys import argv

script, user_name = argv
prompt = '> '

print "Hi %s, I'm the %s script." % (user_name, script)
print "I'd like to ask you a few questions."
print "Do you like me %s?" % user_name
likes = raw_input(prompt)

print "Where do you live %s?" % user_name
lives = raw_input(prompt)

print "What kind of computer do you have?"
computer = raw_input(prompt)

print """
Alright, so you said %r about liking me.
You live in %r.  Not sure where that is.
And you have a %r computer.  Nice.
""" % (likes, lives, computer)
```

Notice though that we make a variable prompt that is set to the prompt we want, and we give that to raw_input instead of typing it over and over. Now if we want to make the prompt something else, we just change it in this one spot and rerun the script.

Very handy.

What You Should See

When you run this, remember that you have to give the script your name for the argv arguments.

Exercise 14 Session

```
$ python ex14.py zed
Hi zed, I'm the ex14.py script.
```

```
I'd like to ask you a few questions.
Do you like me zed?
>  Yes
Where do you live zed?
>  San Francisco
What kind of computer do you have?
>  Tandy 1000

Alright, so you said 'Yes' about liking me.
You live in 'San Francisco'.  Not sure where that is.
And you have a 'Tandy 1000' computer.  Nice.
```

Study Drills

1. Find out what Zork and Adventure were. Try to find a copy and play it.
2. Change the `prompt` variable to something else entirely.
3. Add another argument and use it in your script.
4. Make sure you understand how I combined a `"""` style multiline string with the `%` format activator as the last print.

Common Student Questions

I get `SyntaxError: invalid syntax` when I run this script.
Again, you have to run it right on the command line, not inside Python. If you type `python` and then try to type `python ex14.py Zed`, it will fail because you are running *Python inside Python*. Close your window and then just type `python ex14.py Zed`.

I don't understand what you mean by changing the prompt?
See the variable `prompt = '> '`. Change that to have a different value. You know this; it's just a string and you've done 13 exercises making them, so take the time to figure it out.

I get the error `ValueError: need more than 1 value to unpack`.
Remember when I said you need to look at the WYSS section and replicate what I did? You need to do the same thing here and focus on how I type the command in and why I have a command line argument.

Can I use double-quotes for the `prompt` variable?
You totally can. Go ahead and try that.

You have a Tandy computer?
I did when I was little.

I get `NameError: name 'prompt' is not defined` when I run it.
You either spelled the name of the `prompt` variable wrong or forgot that line. Go back and compare each line of code to mine, and start at the bottom of the script and work your way to the top.

How can I run this from IDLE?
Don't use IDLE.

EXERCISE 15

Reading Files

Everything you've learned about raw_input and argv is so you can start reading files. You may have to play with this exercise the most to understand what's going on, so do it carefully and remember your checks. Working with files is an easy way to *erase your work* if you are not careful.

This exercise involves writing two files. One is your usual ex15.py file that you will run, but the *other* is named ex15_sample.txt. This second file isn't a script but a plain text file we'll be reading in our script. Here are the contents of that file:

```
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

What we want to do is "open" that file in our script and print it out. However, we do not want to just "hard code" the name ex15_sample.txt into our script. "Hard coding" means putting some bit of information that should come from the user as a string right in our program. That's bad because we want it to load other files later. The solution is to use argv and raw_input to ask the user what file the user wants instead of "hard coding" the file's name.

ex15.py

```
1    from sys import argv
2
3    script, filename = argv
4
5    txt = open(filename)
6
7    print "Here's your file %r:" % filename
8    print txt.read()
9
10   print "Type the filename again:"
11   file_again = raw_input("> ")
12
13   txt_again = open(file_again)
14
15   print txt_again.read()
```

A few fancy things are going on in this file, so let's break it down real quick:

Lines 1–3 should be a familiar use of argv to get a filename. Next we have line 5 where we use a new command open. Right now, run pydoc open and read the instructions. Notice how like your own scripts and raw_input, it takes a parameter and returns a value you can set to your own variable. You just opened a file.

Line 7 we print a little line, but on line 8 we have something very new and exciting. We call a function on `txt`. What you got back from open is a `file`, and it's also got commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters. Just like with open and `raw_input`. The difference is that when you say `txt.read()` you are saying, "Hey txt! Do your read command with no parameters!"

The remainder of the file is more of the same, but we'll leave the analysis to you in the Study Drills.

What You Should See

I made a file called "ex15_sample.txt" and ran my script.

Exercise 15 Session

```
$ python ex15.py ex15_sample.txt
Here's your file 'ex15_sample.txt':
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.


Type the filename again:
>  ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

Study Drills

This is a big jump, so be sure you do this Study Drill as best you can before moving on.

1. Above each line, write out in English what that line does.
2. If you are not sure, ask someone for help or search online. Many times searching for "python THING" will find answers for what that THING does in Python. Try searching for "python open."
3. I used the name "commands" here, but they are also called "functions" and "methods." Search around online to see what other people do to define these. Do not worry if they confuse you. It's normal for programmers to confuse you with vast extensive knowledge.
4. Get rid of the part from lines 10–15 where you use `raw_input` and try the script then.
5. Use only `raw_input` and try the script that way. Think of why one way of getting the filename would be better than another.

6. Run pydoc `file` and scroll down until you see the `read()` command (method/function). See all the other ones you can use? Skip the ones that have __ (two underscores) in front because those are junk. Try some of the other commands.
7. Start `python` again and use `open` from the prompt. Notice how you can open files and run `read` on them right there?
8. Have your script also do a `close()` on the `txt` and `txt_again` variables. It's important to close files when you are done with them.

Common Student Questions

Does `txt = open(filename)` return the contents of the file?
No, it doesn't. It actually makes something called a "file object." You can think of it like an old tape drive that you saw on mainframe computers in the 1950s or even like a DVD player from today. You can move around inside them, and then "read" them, but the file is not the contents.

I can't type code into my Terminal/PowerShell like you say in Study Drill #7.
First thing, from the command line just type `python` and hit Enter. Now you are in `python` as we've done a few other times. Once you have that you can just type in code and Python will run it in little pieces. Play with that. To get out of it type `quit()` and hit Enter.

What does `from sys import argv` mean?
For now, just understand that `sys` is a package, and this phrase just says to get the `argv` feature from that package. You'll learn more about these later.

I put the name of the file in as `script, ex15_sample.txt = argv` but it doesn't work.
No, that's not how you do it. Make the code exactly like mine, then run it from the command line the exact same way I do. You don't put the names of files in; you let Python put the name in.

Why is there no error when we open the file twice?
Python will not restrict you from opening a file more than once, and in fact sometimes this is necessary.

Reading and Writing Files

If you did the Study Drills from the last exercise, you should have seen all sorts of commands (methods/functions) you can give to files. Here's the list of commands I want you to remember:

- close—Closes the file. Like `File->Save..` in your editor.
- read—Reads the contents of the file. You can assign the result to a variable.
- readline—Reads just one line of a text file.
- truncate—Empties the file. Watch out if you care about the file.
- write(stuff)—Writes stuff to the file.

For now, these are the important commands you need to know. Some of them take parameters, but we do not really care about that. You only need to remember that `write` takes a parameter of a string you want to write to the file.

Let's use some of this to make a simple little text editor:

ex16.py

```python
from sys import argv

script, filename = argv

print "We're going to erase %r." % filename
print "If you don't want that, hit CTRL-C (^C)."
print "If you do want that, hit RETURN."

raw_input("?")

print "Opening the file..."
target = open(filename, 'w')

print "Truncating the file.  Goodbye!"
target.truncate()

print "Now I'm going to ask you for three lines."

line1 = raw_input("line 1: ")
line2 = raw_input("line 2: ")
line3 = raw_input("line 3: ")

print "I'm going to write these to the file."

target.write(line1)
target.write("\n")
```

```
27   target.write(line2)
28   target.write("\n")
29   target.write(line3)
30   target.write("\n")
31
32   print "And finally, we close it."
33   target.close()
```

That's a large file—probably the largest you have typed in. So go slow, do your checks, and make it run. One trick is to get bits of it running at a time. Get lines 1–8 running, then five more, then a few more, and so on, until it's all done and running.

What You Should See

There are actually two things you will see. First the output of your new script:

Exercise 16 Session

```
$ python ex16.py test.txt
We're going to erase 'test.txt'.
If you don't want that, hit CTRL-C (^C).
If you do want that, hit RETURN.
?
Opening the file...
Truncating the file.  Goodbye!
Now I'm going to ask you for three lines.
line 1:  Mary had a little lamb
line 2:  It's fleece was white as snow
line 3:  It was also tasty
I'm going to write these to the file.
And finally, we close it.
```

Now, open up the file you made (in my case `test.txt`) in your editor and check it out. Neat, right?

Study Drills

1. If you feel you do not understand this, go back through and use the comment trick to get it squared away in your mind. One simple English comment above each line will help you understand or at least let you know what you need to research more.
2. Write a script similar to the last exercise that uses `read` and `argv` to read the file you just created.
3. There's too much repetition in this file. Use strings, formats, and escapes to print out `line1`, `line2`, and `line3` with just one `target.write()` command instead of six.

4. Find out why we had to pass a `'w'` as an extra parameter to open. Hint: open tries to be safe by making you explicitly say you want to write a file.

5. If you open the file with `'w'` mode, then do you really need the `target.truncate()`? Go read the docs for Python's open function and see if that's true.

Common Student Questions

Is the `truncate()` necessary with the `'w'` parameter?
See Study Drills #5.

What does `'w'` mean?
It's really just a string with a character in it for the kind of mode for the file. If you use `'w'`, then you're saying "open this file in 'write' mode"—hence the `'w'` character. There's also `'r'` for "read," `'a'` for append, and modifiers on these.

What are the modifiers to the file modes we can use?
The most important one to know for now is the + modifier, so you can do `'w+'`, `'r+'`, and `'a+'`. This will open the file in both read and write mode and, depending on the character used, position the file in different ways.

Does just doing open`(filename)` open it in `'r'` (read) mode?
Yes, that's the default for the open`()` function.

More Files

Now let's do a few more things with files. We're going to actually write a Python script to copy one file to another. It'll be very short but will give you some ideas about other things you can do with files.

ex17.py

```
from sys import argv
from os.path import exists

script, from_file, to_file = argv

print "Copying from %s to %s" % (from_file, to_file)

# we could do these two on one line too, how?
in_file = open(from_file)
indata = in_file.read()

print "The input file is %d bytes long" % len(indata)

print "Does the output file exist? %r" % exists(to_file)
print "Ready, hit RETURN to continue, CTRL-C to abort."
raw_input()

out_file = open(to_file, 'w')
out_file.write(indata)

print "Alright, all done."

out_file.close()
in_file.close()
```

You should immediately notice that we `import` another handy command named `exists`. This returns `True` if a file exists, based on its name in a string as an argument. It returns `False` if not. We'll be using this function in the second half of this book to do lots of things, but right now you should see how you can import it.

Using `import` is a way to get tons of free code other better (well, usually) programmers have written so you do not have to write it.

What You Should See

Just like your other scripts, run this one with two arguments: the file to copy from and the file to copy it to. I'm going to use a simple test file named `test.txt` again:

Exercise 17 Session

```
$ cat test.txt
This is a test file.
$
$ python ex17.py test.txt new_file.txt
Copying from test.txt to new_file.txt
The input file is 21 bytes long
Does the output file exist? False
Ready, hit RETURN to continue, CTRL-C to abort.

Alright, all done.
```

It should work with any file. Try a bunch more and see what happens. Just be careful you do not blast an important file.

WARNING! Did you see that trick I did with `cat` to show the file? You can learn how to do that in the appendix.

Study Drills

1. Go read up on Python's `import` statement, and start `python` to try it out. Try importing some things and see if you can get it right. It's alright if you do not.
2. This script is *really* annoying. There's no need to ask you before doing the copy, and it prints too much out to the screen. Try to make it more friendly to use by removing features.
3. See how short you can make the script. I could make this one line long.
4. Notice at the end of the WYSS I used something called `cat`? It's an old command that "con*cat*enates" files together, but mostly it's just an easy way to print a file to the screen. Type `man cat` to read about it.
5. Windows people, find the alternative to `cat` that Linux/OSX people have. Do not worry about `man` since there is nothing like that.
6. Find out why you had to do `output.close()` in the code.

Common Student Questions

Why is the `'w'` in quotes?
That's a string. You've been using them for a while now, so make sure you know what a string is.

No way you can make this one line!
That ; depends ; on ; how ; you ; define ; one ; line ; of ; code.

What does the `len()` function do?
It gets the length of the string that you pass to it and then returns that as a number. Play with it.

When I try to make this script shorter, I get an error when I close the files at the end.
You probably did something like this, `indata = open(from_file).read()`, which means you don't need to then do `in_file.close()` when you reach the end of the script. It should already be closed by Python once that one line runs.

Is it normal to feel like this exercise was really hard?
Yes, it is totally normal. Programming may not "click" for you until maybe even Exercise 36, or it might not until you finish the book and then make something with Python. Everyone is different, so just keep going and keep reviewing exercises that you had trouble with until it clicks. Be patient.

I get a `Syntax:EOL while scanning string literal` error.
You forgot to end a string properly with a quote. Go look at that line again.

EXERCISE 18

Names, Variables, Code, Functions

Big title, right? I am about to introduce you to *the function*! Dum dum dah! Every programmer will go on and on about functions and all the different ideas about how they work and what they do, but I will give you the simplest explanation you can use right now.

Functions do three things:

1. They name pieces of code the way variables name strings and numbers.
2. They take arguments the way your scripts take `argv`.
3. Using #1 and #2, they let you make your own "mini-scripts" or "tiny commands."

You can create a function by using the word `def` in Python. I'm going to have you make four different functions that work like your scripts, and I'll then show you how each one is related.

ex18.py

```
# this one is like your scripts with argv
def print_two(*args):
    arg1, arg2 = args
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# ok, that *args is actually pointless, we can just do this
def print_two_again(arg1, arg2):
    print "arg1: %r, arg2: %r" % (arg1, arg2)

# this just takes one argument
def print_one(arg1):
    print "arg1: %r" % arg1

# this one takes no arguments
def print_none():
    print "I got nothin'."


print_two("Zed","Shaw")
print_two_again("Zed","Shaw")
print_one("First!")
print_none()
```

Let's break down the first function, `print_two`, which is the most similar to what you already know from making scripts:

1. First we tell Python we want to make a function using `def` for "define."

2. On the same line as `def`, we then give the function a name. In this case, we just called it `print_two`, but it could be `peanuts` too. It doesn't matter, except that your function should have a short name that says what it does.
3. Then we tell it we want `*args` (asterisk args), which is a lot like your `argv` parameter but for functions. This *has* to go inside `()` parentheses to work.
4. Then we end this line with a `:` colon and start indenting.
5. After the colon all the lines that are indented four spaces will become attached to this name, `print_two`. Our first indented line is one that unpacks the arguments the same as with your scripts.
6. To demonstrate how it works, we print these arguments out, just like we would in a script.

Now, the problem with `print_two` is that it's not the easiest way to make a function. In Python we can skip the whole unpacking args and just use the names we want right inside `()`. That's what `print_two_again` does.

After that, you have an example of how you make a function that takes one argument in `print_one`.

Finally you have a function that has no arguments in `print_none`.

WARNING! This is very important. Do *not* get discouraged right now if this doesn't quite make sense. We're going to do a few exercises linking functions to your scripts and show you how to make more. For now just keep thinking "mini-script" when I say "function," and keep playing with them.

What You Should See

If you run the above script, you should see the following:

Exercise 18 Session

```
$ python ex18.py
arg1: 'Zed', arg2: 'Shaw'
arg1: 'Zed', arg2: 'Shaw'
arg1: 'First!'
I got nothin'.
```

Right away you can see how a function works. Notice that you used your functions the way you use things like `exists`, `open`, and other "commands." In fact, I've been tricking you because in Python those "commands" are just functions. This means you can make your own commands and use them in your scripts too.

Study Drills

Write out a `function checklist` for later exercises. Write these on an index card and keep it by you while you complete the rest of these exercises or until you feel you do not need it:

1. Did you start your function definition with `def`?
2. Does your function name have only characters and _ (underscore) characters?
3. Did you put an open parenthesis `(` right after the function name?
4. Did you put your arguments after the parenthesis `(` separated by commas?
5. Did you make each argument unique (meaning no duplicated names)?
6. Did you put a close parenthesis and a colon `):` after the arguments?
7. Did you indent all lines of code you want in the function four spaces? No more, no less.
8. Did you "end" your function by going back to writing with no indent (`dedenting` we call it)?

And when you run ("use" or "call") a function, check these things:

1. Did you call/use/run this function by typing its name?
2. Did you put the `(` character after the name to run it?
3. Did you put the values you want into the parenthesis separated by commas?
4. Did you end the function call with a `)` character?

Use these two checklists on the remaining lessons until you do not need them anymore. Finally, repeat this a few times: "To 'run,' 'call,' or 'use' a function all mean the same thing."

Common Student Questions

What's allowed for a function name?
Just like variable names, anything that doesn't start with a number and is letters, numbers, and underscores will work.

What does the * in *`args` do?
That tells Python to take all the arguments to the function and then put them in args as a list. It's like `argv` that you've been using, but for functions. It's not normally used too often unless specifically needed.

This feels really boring and monotonous.
That's good. It means you're starting to get better at typing in the code and understanding what it does. To make it less boring, take everything I tell you to type in, and then break it on purpose.

Functions and Variables

Functions may have been a mind-blowing amount of information, but do not worry. Just keep doing these exercises and going through your checklist from the last exercise and you will eventually get it.

There is one tiny point though that you might not have realized, which we'll reinforce right now. The variables in your function are not connected to the variables in your script. Here's an exercise to get you thinking about this:

ex19.py

```
def cheese_and_crackers(cheese_count, boxes_of_crackers):
    print "You have %d cheeses!" % cheese_count
    print "You have %d boxes of crackers!" % boxes_of_crackers
    print "Man that's enough for a party!"
    print "Get a blanket.\n"


print "We can just give the function numbers directly:"
cheese_and_crackers(20, 30)


print "OR, we can use variables from our script:"
amount_of_cheese = 10
amount_of_crackers = 50

cheese_and_crackers(amount_of_cheese, amount_of_crackers)


print "We can even do math inside too:"
cheese_and_crackers(10 + 20, 5 + 6)


print "And we can combine the two, variables and math:"
cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

This shows all the different ways we're able to give our function `cheese_and_crackers` the values it needs to print them. We can give it straight numbers. We can give it variables. We can give it math. We can even combine math and variables.

In a way, the arguments to a function are kind of like our = character when we make a variable. In fact, if you can use = to name something, you can usually pass it to a function as an argument.

What You Should See

You should study the output of this script and compare it with what you think you should get for each of the examples in the script.

Exercise 19 Session

```
$ python ex19.py
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.

OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.

We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.

And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

Study Drills

1. Go back through the script and type a comment above each line, explaining in English what it does.
2. Start at the bottom and read each line backward, saying all the important characters.
3. Write at least one more function of your own design, and run it 10 different ways.

Common Student Questions

How can there possibly be 10 different ways to run a function?
Believe it or not, there's a theoretically infinite number of ways to call any function. In this case, do it like I've got with lines 8–12 and be creative.

Is there a way to analyze what this function is doing so I can understand it better?
There's many different ways, but try putting an English comment above each line describing what the line does. Another trick is to read the code out loud. Yet another is to print the code out and draw on the paper with pictures and comments showing what's going on.

What if I want to ask the user for the numbers of cheese and crackers?
Remember, you just need to use `int()` to convert what you get from `raw_input()`.

Does making the variables on lines 13 and 14 change the variables in the function?
Nope, those variables are separate and live outside the function. They are then passed to the function and temporary versions are made just for the function's run. When the function exits, these temporary variables go away and everything keeps working. Keep going in the book and this should become clearer.

Is it bad to have global variables (like on lines 13 and 14) with the same name as function variables?
Yes, since then you're not quite sure which one you're talking about. But sometimes necessity means you have to use the same name, or you might do it on accident. Just avoid it whenever you can.

Are lines 12–19 overwriting the function `cheese_and_crackers`?
No, not at all. It's calling them, which is basically a temporary jump to the first line of the function, then a jump back after the last line of the function has ended. It's not replacing the function with anything.

Is there a limit to the number of arguments a function can have?
It depends on the version of Python and the computer you're on, but it is fairly large. The practical limit, though, is about five arguments before the function becomes annoying to use.

Can you call a function within a function?
Yes, you'll make a game that does this later in the book.

EXERCISE 20

Functions and Files

Remember your checklist for functions, then do this exercise paying close attention to how functions and files can work together to make useful stuff.

ex20.py

```
from sys import argv

script, input_file = argv

def print_all(f):
    print f.read()

def rewind(f):
    f.seek(0)

def print_a_line(line_count, f):
    print line_count, f.readline()

current_file = open(input_file)

print "First let's print the whole file:\n"

print_all(current_file)

print "Now let's rewind, kind of like a tape."

rewind(current_file)

print "Let's print three lines:"

current_line = 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)

current_line = current_line + 1
print_a_line(current_line, current_file)
```

Pay close attention to how we pass in the current line number each time we run `print_a_line`.

What You Should See

Exercise 20 Session

```
$ python ex20.py test.txt
First let's print the whole file:

This is line 1
This is line 2
This is line 3

Now let's rewind, kind of like a tape.
Let's print three lines:
1 This is line 1

2 This is line 2

3 This is line 3
```

Study Drills

1. Go through and write English comments for each line to understand what's going on.
2. Each time `print_a_line` is run, you are passing in a variable `current_line`. Write out what `current_line` is equal to on each function call, and trace how it becomes `line_count` in `print_a_line`.
3. Find each place a function is used, and go check its `def` to make sure that you are giving it the right arguments.
4. Research online what the `seek` function for `file` does. Try `pydoc file` and see if you can figure it out from there.
5. Research the shorthand notation += and rewrite the script to use that.

Common Student Questions

What is `f` in the `print_all` and other functions?
The `f` is a variable just like you had in other functions in Exercise 18, except this time it's a file. A file in Python is kind of like an old tape drive on a mainframe, or maybe a DVD player. It has a "read head," and you can "seek" this read head around the file to positions, then work with it there. Each time you do `f.seek(0)`, you're moving to the start of the file. Each time you do

`f.readline()`, you're reading a line from the file and moving the read head to right after the `\n` that ends that file. This will be explained more as you go on.

Why are there empty lines between the lines in the file?
The `readline()` function returns the `\n` that's in the file at the end of that line. This means that print's `\n` is being added to the one already returned by `readline()`. To change this behavior simply add a `,` (comma) at the end of print so that it doesn't print its own `\n`.

Why does `seek(0)` not set the `current_line` to 0?
First, the `seek()` function is dealing in *bytes*, not lines. So that's going to the 0 byte (first byte) in the file. Second, `current_line` is just a variable and has no real connection to the file at all. We are manually incrementing it.

What is +=?
You know how in English I can rewrite "it is" to be "it's"? Or I can rewrite "you are" to "you're"? That's called a contraction, and this is kind of like a contraction for the two operations = and +. That means `x = x + y` is the same as `x += y`.

How does `readline()` know where each line is?
Inside `readline()` is code that scans each byte of the file until it finds a `\n` character, then stops reading the file to return what it found so far. The file `f` is responsible for maintaining the current position in the file after each `readline()` call, so that it will keep reading each line.

Functions Can Return Something

You have been using the = character to name variables and set them to numbers or strings. We're now going to blow your mind again by showing you how to use = and a new Python word `return` to set variables to be a *value from a function*. There will be one thing to pay close attention to, but first type this in:

ex21.py

```python
def add(a, b):
    print "ADDING %d + %d" % (a, b)
    return a + b

def subtract(a, b):
    print "SUBTRACTING %d - %d" % (a, b)
    return a - b

def multiply(a, b):
    print "MULTIPLYING %d * %d" % (a, b)
    return a * b

def divide(a, b):
    print "DIVIDING %d / %d" % (a, b)
    return a / b


print "Let's do some math with just functions!"

age = add(30, 5)
height = subtract(78, 4)
weight = multiply(90, 2)
iq = divide(100, 2)

print "Age: %d, Height: %d, Weight: %d, IQ: %d" % (age, height, weight, iq)


# A puzzle for the extra credit, type it in anyway.
print "Here is a puzzle."

what = add(age, subtract(height, multiply(weight, divide(iq, 2))))

print "That becomes: ", what, "Can you do it by hand?"
```

We are now doing our own math functions for `add`, `subtract`, `multiply`, and `divide`. The important thing to notice is the last line where we say `return a + b` (in add). What this does is the following:

1. Our function is called with two arguments: a and b.
2. We print out what our function is doing, in this case `ADDING`.
3. Then we tell Python to do something kind of backward: we return the addition of `a + b`. You might say this as, "I add a and b, then return them."
4. Python adds the two numbers. Then when the function ends, any line that runs it will be able to assign this `a + b` result to a variable.

As with many other things in this book, you should take this real slow, break it down, and try to trace what's going on. To help there's extra credit to get you to solve a puzzle and learn something cool.

What You Should See

Exercise 21 Session

```
$ python ex21.py
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50
Here is a puzzle.
DIVIDING 50 / 2
MULTIPLYING 180 * 25
SUBTRACTING 74 - 4500
ADDING 35 + -4426
That becomes:  -4391 Can you do it by hand?
```

Study Drills

1. If you aren't really sure what `return` does, try writing a few of your own functions and have them return some values. You can return anything that you can put to the right of an =.
2. At the end of the script is a puzzle. I'm taking the return value of one function and *using* it as the argument of another function. I'm doing this in a chain so that I'm kind of creating a formula using the functions. It looks really weird, but if you run the script, you can see the results. What you should do is try to figure out the normal formula that would recreate this same set of operations.

3. Once you have the formula worked out for the puzzle, get in there and see what happens when you modify the parts of the functions. Try to change it on purpose to make another value.
4. Finally, do the inverse. Write out a simple formula and use the functions in the same way to calculate it.

This exercise might really whack your brain out, but take it slow and easy and treat it like a little game. Figuring out puzzles like this is what makes programming fun, so I'll be giving you more little problems like this as we go.

Common Student Questions

Why does Python print the formula or the functions "backward"?
It's not really backward; it's "inside out." When you start breaking down the function into separate formulas and function calls, you'll see how it works. Try to understand what I mean by "inside out" rather than "backward."

How can I use `raw_input()` to enter my own values?
Remember `int(raw_input())`? The problem with that is then you can't enter floating point, so also try using `float(raw_input())` instead.

What do you mean by "write out a formula"?
Try `24 + 34 / 100 - 1023` as a start. Convert that to use the functions. Now come up with your own similar math equation and use variables so it's more like a formula.

What Do You Know So Far?

There won't be any code in this exercise or the next one, so there's no WYSS or Study Drills either. In fact, this exercise is like one giant Study Drills section. I'm going to have you do a form of review of what you have learned so far.

First, go back through every exercise you have done so far and write down every word and symbol (another name for "character") that you have used. Make sure your list of symbols is complete.

Next to each word or symbol, write its name and what it does. If you can't find a name for a symbol in this book, then look for it online. If you do not know what a word or symbol does, then go read about it again and try using it in some code.

You may run into a few things you just can't find out or know, so just keep those on the list and be ready to look them up when you find them.

Once you have your list, spend a few days rewriting the list and double-checking that it's correct. This may get boring, but push through and really nail it down.

Once you have memorized the list and what they do, then you should step it up by writing out tables of symbols, their names, and what they do *from memory*. When you hit some you can't recall from memory, go back and memorize them again.

WARNING! The most important thing when doing this exercise is: "There is no failure, only trying."

What You Are Learning

It's important when you are doing a boring, mindless memorization exercise like this to know why. It helps you focus on a goal and know the purpose of all your efforts.

In this exercise, you are learning the names of symbols so that you can read source code more easily. It's similar to learning the alphabet and basic words of English, except this Python alphabet has extra symbols you might not know.

Just take it slow and do not hurt your brain. Hopefully by now these symbols are natural for you, so this isn't a big effort. It's best to take 15 minutes at a time with your list and then take a break. Giving your brain a rest will help you learn faster with less frustration.

EXERCISE 23

Read Some Code

You should have spent the last week getting your list of symbols straight and locked in your mind. Now you get to apply this to another week of reading code on the internet. This exercise will be daunting at first. I'm going to throw you in the deep end for a few days and have you just try your best to read and understand some source code from real projects. The goal isn't to get you to understand code, but to teach you the following three skills:

1. Finding Python source code for things you need.
2. Reading through the code and looking for files.
3. Trying to understand code you find.

At your level, you really do not have the skills to evaluate the things you find, but you can benefit from getting exposure and seeing how things look.

When you do this exercise, think of yourself as an anthropologist, trucking through a new land with just barely enough of the local language to get around and survive. Except, of course, that you will actually get out alive because the internet isn't a jungle.

Here's what you do:

1. Go to bitbucket.org, github.com, or gitorious.org with your favorite web browser and search for "python."
2. Avoid any project that mentions "Python 3." That'll only confuse you.
3. Pick a random project and click on it.
4. Click on the `Source` tab and browse through the list of files and directories until you find a .py file (but not setup.py—that's useless).
5. Start at the top and read through it, taking notes on what you think it does.
6. If any symbols or strange words seem to interest you, write them down to research later.

That's it. Your job is to use what you know so far and see if you can read the code and get a grasp of what it does. Try skimming the code first, and then read it in detail. Maybe also try to take very difficult parts and read each symbol you know out loud.

Now try some of these other sites:

- launchpad.net
- sourceforge.net
- freecode.com

EXERCISE 24

More Practice

You are getting to the end of this section. You should have enough Python "under your fingers" to move on to learning about how programming really works, but you should do some more practice. This exercise is longer and all about building up stamina. The next exercise will be similar. Do them, get them exactly right, and do your checks.

ex24.py

```python
print "Let's practice everything."
print 'You\'d need to know \'bout escapes with \\ that do \n newlines and \t tabs.'

poem = """
\tThe lovely world
with logic so firmly planted
cannot discern \n the needs of love
nor comprehend passion from intuition
and requires an explanation
\n\t\twhere there is none.
"""

print "--------------"
print poem
print "--------------"


five = 10 - 2 + 3 - 6
print "This should be five: %s" % five

def secret_formula(started):
    jelly_beans = started * 500
    jars = jelly_beans / 1000
    crates = jars / 100
    return jelly_beans, jars, crates


start_point = 10000
beans, jars, crates = secret_formula(start_point)

print "With a starting point of: %d" % start_point
print "We'd have %d beans, %d jars, and %d crates." % (beans, jars, crates)

start_point = start_point / 10

print "We can also do that this way:"
print "We'd have %d beans, %d jars, and %d crates." % secret_formula(start_point)
```

What You Should See

Exercise 24 Session

```
$ python ex24.py
Let's practice everything.
You'd need to know 'bout escapes with \ that do
 newlines and       tabs.
--------------

       The lovely world
with logic so firmly planted
cannot discern
 the needs of love
nor comprehend passion from intuition
and requires an explanation

              where there is none.

--------------
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000 jars, and 50 crates.
We can also do that this way:
We'd have 500000 beans, 500 jars, and 5 crates.
```

Study Drills

1. Make sure to do your checks: read it backward, read it out loud, and put comments above confusing parts.
2. Break the file on purpose, then run it to see what kinds of errors you get. Make sure you can fix it.

Common Student Questions

How come you call the variable `jelly_beans` but the name beans later?
That's part of how a function works. Remember that inside the function the variable is temporary, and when you return it, then it can be assigned to a variable for later. I'm just making a new variable named beans to hold the return value.

What do you mean by reading the code backward?
Start at the last line. Compare that line in your file to the same line in mine. Once it's exactly the same, move up to the next line. Do this until you get to the first line of the file.

Who wrote that poem?
I did. Not all my poems suck.

Even More Practice

We're going to do some more practice involving functions and variables to make sure you know them well. This exercise should be straightforward for you to type in, break down, and understand.

However, this exercise is a little different. You won't be running it. Instead *you* will import it into Python and run the functions yourself.

ex25.py

```python
def break_words(stuff):
    """This function will break up words for us."""
    words = stuff.split(' ')
    return words

def sort_words(words):
    """Sorts the words."""
    return sorted(words)

def print_first_word(words):
    """Prints the first word after popping it off."""
    word = words.pop(0)
    print word

def print_last_word(words):
    """Prints the last word after popping it off."""
    word = words.pop(-1)
    print word

def sort_sentence(sentence):
    """Takes in a full sentence and returns the sorted words."""
    words = break_words(sentence)
    return sort_words(words)

def print_first_and_last(sentence):
    """Prints the first and last words of the sentence."""
    words = break_words(sentence)
    print_first_word(words)
    print_last_word(words)

def print_first_and_last_sorted(sentence):
    """Sorts the words then prints the first and last one."""
    words = sort_sentence(sentence)
    print_first_word(words)
    print_last_word(words)
```

First, run this like normal with `python ex25.py` to find any errors you have made. Once you have found all the errors you can and fixed them, you will then want to follow the WYSS section to complete the exercise.

What You Should See

In this exercise, we're going to interact with your `.py` file inside the `python` interpreter you used periodically to do calculations. You run that from the shell like this:

```
$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Yours will look a little different from mine, but once you see the >>> prompt you can then type Python code in and it will run immediately.

Here's what it looks like when I do it:

Exercise 25 Python Session

```
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import ex25
>>> sentence = "All good things come to those who wait."
>>> words = ex25.break_words(sentence)
>>> words
['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.']
>>> sorted_words = ex25.sort_words(words)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_word(words)
All
>>> ex25.print_last_word(words)
wait.
>>> wrods
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'wrods' is not defined
>>> words
['good', 'things', 'come', 'to', 'those', 'who']
>>> ex25.print_first_word(sorted_words)
All
>>> ex25.print_last_word(sorted_words)
who
>>> sorted_words
```

```
['come', 'good', 'things', 'those', 'to', 'wait.']
>>> sorted_words = ex25.sort_sentence(sentence)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_and_last(sentence)
All
wait.
>>> ex25.print_first_and_last_sorted(sentence)
All
who
```

Let's break this down line by line to make sure you know what's going on:

- **Line 5.** You import *your* ex25.py Python file, just like other imports you have done. Notice you do not need to put the .py at the end to import it. When you do this, you make a module that has all your functions in it to use.
- **Line 6.** You made a sentence to work with.
- **Line 7.** You use the ex25 module and call your first function ex25.break_words. The . (dot, period) symbol is how you tell Python, "Hey, inside ex25 there's a function called break_words and I want to run it."
- **Line 8.** We just type words, and Python will print out what's in that variable (line 9). It looks weird, but this is a list that you will learn about later.
- **Lines 10–11.** We do the same thing with ex25.sort_words to get a sorted sentence.
- **Lines 13–16.** We use ex25.print_first_word and ex25.print_last_word to get the first and last word printed out.
- **Line 17.** This is interesting. I made a mistake and typed the words variable as wrods so Python gave me an error on lines 18–20.
- **Lines 21–22.** We print the modified words list. Notice that since we printed the first and last one, those words are now missing.

The remaining lines are for you to figure out and analyze in the Study Drills.

Study Drills

1. Take the remaining lines of the WYSS output and figure out what they are doing. Make sure you understand how you are running your functions in the ex25 module.
2. Try doing this: help(ex25) and also help(ex25.break_words). Notice how you get help for your module and how the help is those odd """ strings you put after each func-