Learn Python The Hard Way

Release 0.2

Zed A. Shaw

May 15, 2010

CONTENTS

Contents:

The Hard Way Is Easier

This simple book is meant to give you a first start in programming. The title says it is the hard way to learn to write code but it's actually not. It's the "hard" way only in that it's the way people used to teach things. In this book you will do something incredibly simple that all programmers actually do to learn a 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 1-2 hours a night, then you'll 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 basic essential skills that a beginning programmer needs to know: Reading And Writing, Attention To Detail, Spotting Differences.

Reading And Writing

It seems stupidly obvious, but if you have a problem typing you'll have a problem learning to code. Not only that, but if you have a problem typing the fairly odd characters in source code then you'll 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 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 an attention to the tiniest details of your work you'll miss key important 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 exacting details of what you are doing.

Spotting Differences

A very important skill that 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. In fact, programmers have invented tools to make this even easier. Of course, the invention of these tools also means that even with years of training finding little differences in programming is difficult.

While you do these exercises and type each one in, you'll be making mistakes. It's inevitable, and even seasoned programmers would make a few mistakes. Your job is to compare what you've written to what's required and fix all the differences, and by doing this you'll train yourself to notice your mistakes, bugs, and other problems caused by them.

Do Not Copy-Paste

The last thing I'll say before we begin is that 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 in how to read, write, and see code. If you copy-paste you are cheating yourself out of the effectiveness of the lessons.

License

This book is Copyright (C) 2010 by Zed A. Shaw. You are free to distribute this book to anyone you want so long as you do *not* charge anything for it *and* it is not altered. You must give away the book in its entirety or not at all.

Exercise 0: The Setup

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

Mac OSX

To complete this exercise you have to finish the following tasks:

1. Go to http://learnpythonthehardway.org/wiki/ExerciseZero with your browser, get the `gedit` text editor, and install it.
2. **Put gedit (your editor) in your Dock so you can reach it easily.**
    (a) Run gedit so we can fix some stupid defaults it has.
    (b) Open `Preferences` from the `gedit menu` and select the `Editor` tab.
    (c) Change `Tab width:` to 4.
    (d) Check off `Insert spaces instead of tabs`.
3. Find your "Terminal" program. Search for it. You'll 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 their name and hitting RETURN.
7. Hit CTRL-D (^D) 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. Search online for help.
10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. Typically you will make the file and then "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. Search online for how to list a directory.

OSX: What You Should See

Here's me doing the above 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. Notice I use a text editor called `vim`. You shouldn't use it, it's too hard to use for you right now.

```
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
mystuff $ vim test.txt
mystuff $ ls
test.txt
mystuff $
```

Windows

Note: Contributed by zhmark.

1. Go to http://learnpythonthehardway.org/wiki/ExerciseZero with your browser, get the `gedit` text editor, and install it. You do not need to be administrator to do that.
2. **Make sure you can get to `gedit` easily by putting it on your desktop and/or in `Quick Launch` - bouth options are availa**
   (a) Run gedit so we can fix some stupid defaults it has.
   (b) Open `Edit->Preferences` select the `Editor` tab.
   (c) Change `Tab width:` to 4.
   (d) Check off `Insert spaces instead of tabs`.
3. Find your "Terminal" program. It's called `Command Prompt`, alternatively just run `cmd`.
4. You may 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 their name and hitting RETURN. a. If you run `python` and it's not there (`python is not recognized.`.) - install it. *Make sure you install Python 2 not Python 3.* b. You may be better off with ActiveState python especially when you miss Administrative rights
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. Search online for help.
10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. Typically you will make the file and then "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. Search online for how to list a directory.

Windows: What You Should See

```
C:\Documents and Settings\you>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


C:\Documents and Settings\you>mkdir mystuff

C:\Documents and Settings\you>cd mystuff

C:\Documents and Settings\you\mystuff>u:\project\gedit\bin\gedit.exe test.txt

C:\Documents and Settings\you\mystuff>
   <bunch of unimportant errors if you istalled it as non-admin - ignore them - hit Enter>
C:\Documents and Settings\you\mystuff>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

C:\Documents and Settings\you\mystuff>
```

You will probably see a very different prompt, Python information, and other stuff but this is the general idea. If your system is different let us know at http://learnpythonthehardway.org and we'll fix it.

Linux

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

1. Go to http://learnpythonthehardway.org/wiki/ExerciseZero with your browser, get the `gedit` text editor, and install it.
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` select the `Editor` tab.
   (c) Change `Tab width:` to 4.
   (d) Check off `Insert spaces instead of tabs`.
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 their name and hitting RETURN. a. If you run `python` and it's not there install it. *Make sure you install Python 2 not Python 3.*
7. Hit CTRL-D (^D) 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. Search online for help.
10. Learn how to change into a directory in the Terminal. Again search online.
11. Use your editor to create a file in this directory. Typically you will make the file and then "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. Search online for how to list a directory.

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
[mystuff]$ vim test.txt
[mystuff]$ ls
test.txt
[mystuff]$
```

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

Warnings For Beginners

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

If a programmer tells you to use `vim` or `emacs` tell them 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 because it is simple and the same on all computers. Professional programmers use gedit so it's good enough for you starting out.

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

A programmer will eventually tell you to use Mac OSX or Linux. If the programmer likes fonts and typography they'll tell you to get a Mac OSX computer. If they like control and have a huge beard then they'll tell you to install Linux. Again, use whatever computer you have right now that works. All you need is `gedit`, a Terminal, and `python`.

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

1. *Write* exercises using gedit.

2. *Run* the exercises you wrote.
3. *Fix* them when they're 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 don't go on, you'll 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.

```
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.'
```

Take the above and type it into a single file named `ex1.py`. This is important as python works best with files ending in `.py`.

> **Warning:** Do not type the numbers on the far left of these lines. Those are called "line numbers" and they're used by programmers to talk about what part of a program is wrong. Python will tell you errors related to these line numbers, but *you* do not type them in.

Then in Terminal you *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, then you've done something wrong. No, the computer is not wrong.

What You Should See

```
$ 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.
$
```

You may see the name of your directory before the `$` which is fine, but if your output is not exactly the same, then find out why and fix it.

If you have an error it will look like this:

```
$ python ex1.py
  File "ex1.py", line 3
    print "I like typing this.
                              ^
SyntaxError: EOL while scanning single-quoted string
$
```

It's important you be able to read these since you'll 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'll find someone else who's had that error and you can probably figure out how to fix it.

Extra Credit

You will also have extra credit you should do to make sure you understand each exercise. 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 for some reason. Each time there is code you should put in a new file, the output you should see when you run the file in terminal, and extra credit you should do.

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

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 do comments.

```
# 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."
```

Warning: This book uses "syntax highlighting", which means that the code looks like it has colors for different symbols and letters. This helps you read it so that you can see some errors, and your editor should use similar coloring, but maybe not exactly the same.

What You Should See

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

Extra Credit

1. Find out if you were right about what the # character does and make sure you know what it's called (octothorpe character).
2. Take your `ex2.py` file and review each line going backwards. 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 above out loud, including saying each character by its name. Did you find more mistakes? Fix them.

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

Exercise 3: Numbers And Math

Every programming language has some kind of way of doing numbers and math. Don't 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 so let's name them right away so you know what they're called. As you type this one in, say the names. When saying them feels boring you can stop saying them. Here's 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 you are to go back and figure out what each of these does and complete the table. For example, + does addition.

```
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
```

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

What You Should See

```
$ 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
$
```

Extra Credit

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 means.
5. Rewrite `ex3.py` to use floating point numbers so it's more accurate.

Exercise 4: Variables And Names

You can print things out 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 programmers have a lousy ability to remember things. If they didn't use good names for things in their software they'd get lost when they came back and tried to read their code again.

If you get stuck with this exercise, remember the tricks you've 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 backwards.
3. Read your `.py` file out loud saying even the characters.

```
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 don't already know. We use this character a lot to put an imaginary space between words in variable names.

What You Should See

```
$ 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.
$
```

Extra Credit

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 extra credit:

1. Explain why the 4.0 is used instead of just 4.
2. Remember that 4.0 is a "floating point" number, make sure you 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. You should also know 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`.

Exercise 5: More Variables And Printing

We'll now do even more typing of variables and printing them out. This time though we'll use something called a "format string". You might not know it, but every time you put " (double-quotes) around a piece of text you've 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'll 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 don't understand it and make it exactly the same.

```
1  my_name = 'Zed A. Shaw'
2  my_age = 35 # not a lie
3  my_height = 74 # inches
4  my_weight = 180 # lbs
5  my_eyes = 'Blue'
6  my_teeth = 'White'
7  my_hair = 'Brown'
8
9  print "Let's talk about %s." % my_name
10 print "He's %d inches tall." % my_height
11 print "He's %d pounds heavy." % my_weight
12 print "Actually that's not too heavy."
13 print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
14 print "His teeth are usually %s depending on the coffee." % my_teeth
15
16 # this line is tricky, try to get it exactly right
17 print "If I add %d, %d, and %d I get %d." % (
18     my_age, my_height, my_weight, my_age + my_height + my_weight)
```

What You Should See

```
$ 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.
$
```

Extra Credit

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 of the Python format characters.
4. Try to write some variables that convert the inches and pounds to centimeters and kilos. Don't just type in the measurements, but work out the math in Python.

Exercise 6: Strings And Text

You have already been writing strings but haven't really known what they do. In this exercise we create a bunch of variables with complex strings so you can see what they're 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` where you would put text you want to go to the string inside `"` or `'` after the `print`. Then Python prints it.

Strings also can contain the format characters you've 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, then you need to put them inside `( )` (parenthesis) 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 can say, "(milk, eggs, bread, soup)" and be done with it.

We will now type in a whole bunch of strings, variables, 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 lets get you started being able to read and write them early on.

```
1  x = "There are %d types of people." % 10
2  binary = "binary"
3  do_not = "don't"
4  y = "Those who know %s and those who %s." % (binary, do_not)
5
6  print x
7  print y
8
9  print "I said: %r." % x
10 print "I also said: '%s'." % y
11
12 hilarious = False
13 joke_evaluation = "Isn't that joke so funny?! %r"
14
15 print joke_evaluation % hilarious
16
17 w = "This is the left side of..."
18 e = "a string with a right side."
19
20 print w + e
```

What You Should See

```
$ 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.
$
```

Extra Credit

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 4 places.
3. Are you sure there's only 4 places? How do you know? Maybe I like lying.
4. Try to explain why adding the two string `w` and `e` with + makes a longer string.

Exercise 7: More Printing

We are now going to do a bunch of exercises where you just type code in and make them run. There won't be much talking since it's just more of the same. The purpose is to build up your chops. See you in a few exercises, and *don't skip!* Don't *paste!*

```
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

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

Extra Credit

For these next few exercises you will have the exact same extra credit.

1. Go back through and write a comment on what each line does.
2. Read each one backwards 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're perfect and never wrong, but it's all an act. They make mistakes all the time.

Exercise 8: Printing, Printing

```
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

```
$ 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.'
$
```

Extra Credit

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

Exercise 9: Printing, Printing, Printing

```
# 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's the days: ", days
print "Here's 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.
"""
```

What You Should See

```
$ python ex9.py
Here's the days:  Mon Tue Wed Thu Fri Sat Sun
Here's 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.

$
```

Extra Credit

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

Exercise 10: What Was That?

In Exercise 9 I threw you some new stuff you haven't seen yet, just to keep you on your toes. In that exercise I showed you two ways to make a string that goes across multiple lines. In the first way, I put the characters `\n` (back-slash `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 `\` (back-slash) character is a way we can put difficult to type characters into a string. There's plenty of these "escape sequences" available for different characters you might want to put in, but there's a special one, the `double back-slash` which is just two of them `\\`. These two characters will print just one back-slash. 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 if you have a string that uses double-quotes and you want to put a double-quote in for the output. If you just put one there then it'd end the string and Python would get confused. Instead you escape it and Python knows to include in the string. Here's an example:

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

```
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.

```
$ 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

$
```

Extra Credit

1. Go search online to see what other escape sequences are available.
2. Try using `"'` (triple-single-quote) instead. Can you see why you might use that instead of `"""`?
3. Try to 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?

Exercise 11: Asking Questions

It's now time to pick up the pace a bit. I've 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 you getting data into your programs. This though is a little tricky so we have to have you learn to do two things that may not make sense right away, but if you stick with it they should click suddenly a few exercises from now.

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've only been printing things and the basics of changing it, but you haven't been able to get any input in. 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. Next exercise we'll do more to explain it.

```
print "How old are you?",
age = raw_input()
print "How many tall are you?",
height = raw_input()
print "How much do you weight?",
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 newline and go to the next line.

What You Should See

```
$ python ex11.py
How old are you? 35
How many tall are you? 6'2"
How much do you weight? 180lbs
So, you're '35' old, '6\'2"' tall and '180lbs' heavy.
$
```

Extra Credit

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?

Exercise 12: Prompting People

When you typed `raw_input()` you were typing the `(` and `)` characters which are parenthesis. 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. You put a string that you want for the prompt inside the `()` so that it looks like this:

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

Which prompts the user with "Name?" and puts the result into the variable `y`.

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

```
1 age = raw_input("How old are you? ")
2 height = raw_input("How many tall are you? ")
3 weight = raw_input("How much do you weight? ")
4
5 print "So, you're %r old, %r tall and %r heavy." % (
6     age, height, weight)
```

What You Should See

```
$ python ex12.py
How old are you? 35
How many tall are you? 6'2"
How much do you weight? 180lbs
So, you're '35' old, '6\'2"' tall and '180lbs' heavy.
$
```

Extra Credit

1. In Terminal where you normally run `python` to run your scripts, type: `pydoc raw_input`. Read what it says.
2. Go look online for what the `pydoc` command does.
3. Do pydoc for `open`, `file`, `os`, and `sys`. It's alright if you don't understand those, just read through and take notes about interesting things.

Exercise 13: Parameters, Unpacking, Variables

We're slowly building up your first few input operations, so we want to 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 a "argument". What we'll do now is write a script that also accepts arguments.

Go ahead and type this program and then I'll explain it in detail:

```
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" and it's actually a very standard name in programming. You'll find it used in many other languages. What this variable does is *hold* the arguments you pass to your Python script when you run it. In the exercises you'll 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 of these variables on the left in order."

After that we just print them out like normal.

What You Should See

When you run the program run it like this:

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

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

```
Your third variable is: 3rd

$ python ex13.py cheese apples bread
The script is called: ex/ex13.py
Your first variable is: cheese
Your second variable is: apples
Your third variable is: bread

$ python ex13.py Zed A. Shaw
The script is called: ex/ex13.py
Your first variable is: Zed
Your second variable is: A.
Your third variable is: Shaw
```

You can actually replace "first", "2nd", and "3rd" with any three things.

Extra Credit

1. Try giving less than three arguments to your script. See that error you get? See if you can explain it.
2. Write a script that has less 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 does more input from a user.

Exercise 14: Prompting And Passing

Let's do one exercise to put using `argv` and `raw_input` together to ask the user something specific. You'll 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 pring a simple > prompt. Similar to in a game like Zork or Adventure.

```
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 this prompt we want, and then 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 have to change it in this one spot and rerun the script.

Very handy.

What You Should See

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

```
$ 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?
> America
```

```
What kind of computer do you have?
> Tandy

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

Extra Credit

1. Go find out what Zork was, and what Adventure was. See if you can 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 multi-line string with the `%` format activator as the last print.

Exercise 15: Reading Files

Everything about `raw_input` and `argv` has been so you can start reading files. This exercise will probably be the one you have to play with the most to understand what's going on, so do it carefully and remember your checks. Working with files is one way to very quickly erase your work if you're not careful.

First, this exercise involves writing two files. One is your usual `ex15.py` file that you'll run, but the *other* is named `ex15_sample.txt`. This second file isn't a script, but just a plain text file we'll be reading in our script. Here's 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 don't want to just "hard code" the name `ex15_sample.txt` into our script. The term "hard coding" means that we've put 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 they want instead of "hard coding" the file's name.

```
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 "I'll also ask you to type it again:"
11 file_again = raw_input("> ")
12
13 txt_again = open(file_again)
14
15 print txt_again.read()
```

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

Line 1-3 should be familiar use of `argv` to get a filename. Next we have line 5 where we use a new command `open`. Right now, go 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`. You see, 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 and parameters. Just like with `open` and `raw_input`. The difference is that when you say `txt.read()` you're 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 extra credit.

What You Should See

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

```
$ 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.


I'll also ask you to type it 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.


$
```

Extra Credit

This is a big jump so you want to make sure you do this extra credit as best you can before moving on.

1. Above each line write out in English what that line does.
2. If you're not sure ask someone for help or search online. Many times searching for "python THING" will find answers for what THING does in python. Try searching for "python open".
3. I used the name "commands" here, but they're also called "functions" and "methods". Search around online to see what other people do to define these. Don't worry if they confuse you, it's normal for a programmer to confuse you with their vast extensive knowledge.
4. Get rid of the part from line 10-16 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. Startup `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're done with them.

Exercise 16: Reading And Writing Files

If you did the extra credit 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 don't 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:

```
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")
target.write(line2)
target.write("\n")
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've 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 5 more, then a few more, etc. until it's all done and running.

What You Should See

There are actually two things you'll see, first the output of your new script:

```
$ 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: To all the people out there.
line 2: I say I don't like my hair.
line 3: I need to shave it off.
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?

Extra Credit

1. If you feel you don't 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 6.
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.

Exercise 17: 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.

```
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?
input = open(from_file)
indata = input.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()

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

print "Alright, all done."
```

You should right away notice that we `import` another handy command named `exists`. This returns `True` if a file exists, based on it's 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 don't 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. If we use your `test.txt` file from before we get this:

```
$ python ex17.py test.txt copied.txt
Copying from test.txt to copied.txt
The input file is 81 bytes long
Does the output file exist? False
Ready, hit RETURN to continue, CTRL-C to abort.
```

```
Alright, all done.

$ cat copied.txt
To all the people out there.
I say I don't like my hair.
I need to shave it off.
$
```

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

> **Warning:** Did you see that trick I did with `cat`?

Extra Credit

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 don't.
2. This script is *really* annoying. There's no need to ask you before doing the copy, 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 1 line long.
4. Notice at the end of the WYSS I used something called *cat*? It's an old comman 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.

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".

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

```
# 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. Still 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.

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 `()` parenthesis to work.
4. Then we end this line with a `:` colon, and start indenting.
5. After the colon all the lines that are indented 4 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. And 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 functiont that has no arguments in `print_none`.

Warning: This is very important, don't 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:

```
$ 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 we did with our functions what you have been doing with things like `exists`, `open`, and other "commands". In fact, I've been tricking you because in Python those "commands" are just functions. This means that you can make your own commands and use them in your scripts too.

Extra Credit

You need to write out a `function checklist` for later exercises. Write these out onto an index card and keep it by you while you complete the rest of these exercises or until you feel you don't 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 indent all lines of code you want in the function 4 spaces? No more, no less.
7. Did you "end" your function by going back to writing with no indent (dedenting we call it)?

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

1. Did you call/use/run this function by typing its name?

2. Did you put（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.

You will use these two checklists on the remaining lessons until you don't need them anymore.

Finally, repeat this a few times:

"To 'run', 'call', or 'use' a function all mean the same thing."

Exercise 19: Functions And Variables

Functions may have been a mind blowing amount of information, but don't worry just keep doing these exercises and going through your checklist from the last exercise and you'll 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:

```
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 to 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 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 then 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 to what you think you should get for each of the examples in the script.

```
$ 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 to 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.
$
```

Extra Credit

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 backwards, saying all the important characters.
3. Write at least one more function of your own design, and run it 10 different ways.

Exercise 20: Functions And Files

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

```
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

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

To all the people out there.
I say I don't like my hair.
I need to shave it off.

Now let's rewind, kind of like a tape.
Let's print three lines:
1 To all the people out there.

2 I say I don't like my hair.

3 I need to shave it off.

$
```

Extra Credit

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`.

1. Find each place a function is used, and go check it's `def` to make sure that you are giving it the right arguments.
2. Research online what the `seek` function for `file` does. Try `pydoc file` and see if you can figure it out from there.
3. Research the shorthand notation += and rewrite the script to use that.

Exercise 21: Functions Can Return

You've 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:

```
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 backwards, 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, and 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 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

```
$ 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?
$
```

Extra Credit

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 functios. It looks really weird, but if you ran 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. First do it with variables then with functions called as arguments to functions.

This exercise might really whack your brain out, but take it slow and easy and treat it like a little game. Figuring things like this out is almost all of the fun there is in programming, so I'll be giving you more little problems like this as we go.

Exercise 22: 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 Extra Credit either. In fact, this exercise is like one giant Extra Credit. I'm going to have you do a form of review on what you should know so far in order to help lock in what you've learned so far.

First, you should go back through every exercise you've done so far and write down every word and symbol (another name for 'character') that you've used. Make sure your list of symbols is complete and you don't miss any.

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

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.

If you think you've memorized the list and what they do, then you should step it up and do this:

1. Take a blank sheet of paper and try to write down as many symbols and words as you can from memory.
2. Compare what you could recall from memory to what was listed on your sheet, fill in the rest.

3. Write down the names from memory of all the symbols you can. 3. Again go look at your list and compare them, correcting any and filling in the remaining

   ones you didn't remember.

1. Finally do the same for what each symbol does: write them from memory and then fill in the ones you don't remember or get wrong using your sheet.

> **Warning:** The most important thing when doing this exercise is: "There is no failure, only trying."

What You're Learning

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

In this exercise you're learning a vocabulary. You're learning the names of symbols so that you can read source code more easily than if you didn't know them. It's similar to learning the alphabet, except the alphabet has extra symbols you might not know.

The technique you're also learning will help you memorize things. A good trick is to try to recall what you can from memory, and then fill in the rest of the ones you don't know to then memorize again. Doing it this way you develop

memory skills that don't require you to have a "trigger" to remember what something is and makes it a more natural piece of knowledge.

Finally, just take it slow and don't 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 it faster with less frustration.

Exercise 23: Read Some Code

You should have spent last week getting your list of symbols straight and get them locked into your mind. Now you get to apply this to another week 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 don't have the skills to evaluate the things you find, but you can benefit from getting exposure to it 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 see survive. Except, of course, that you'll actually get out alive because the internet isn't a jungle. Anyway.

Here's what you do:

1. Go to bitbucket.org and search for "python".
2. Avoid any project with "Python 3" mentioned. That'll only confuse you.
3. Take a random project you find, 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 to see if you can read the code and get a grasp of what it does. You should try to do it first by skimming, and then by reading in detail anything you find. Maybe also try taking very difficult parts and reading each symbol you know outloud.

You should then try several three other sites:

- github.com
- launchpad.net
- koders.com

On each of these sites you may find weird files ending in `.c` so stick to `.py` files like the ones you have written in this book.

A final fun thing to do is use the above four sources of Python code and type in topics you're interested in instead of "python". Search for "journalism", "cooking", "physics", or anything you're curious about. Chances are there's some code out there that someone wrote you could use right away.

Exercise 24: More Practice

We're coming to the end of this section, where should have enough of Python "under your fingers" to move onto 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.

```
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 explantion
\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

```
$ 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 explantion

                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.
$
```

Extra Credit

1. Make sure to do your checks: read it backwards, read it out loud, 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.

Exercise 25: 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 straight forward for you to type in and you should be able to break it down and understand it.

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

```
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 thing you should do is run this like normal with `python ex25.py` to find any errors you've made. Once you've found all of the errors you can this way and fixed them, you'll 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.

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

```
1  $ python
2  Python 2.5.1 (r251:54863, Feb  6 2009, 19:02:12)
3  [GCC 4.0.1 (Apple Inc. build 5465)] on darwin
4  Type "help", "copyright", "credits" or "license" for more information.
5  >>> import ex25
6  >>> sentence = "All good things come to those who wait."
7  >>> words = ex25.break_words(sentence)
8  >>> words
9  ['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.']
10 >>> sorted_words = ex25.sort_words(words)
11 >>> sorted_words
12 ['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
13 >>> ex25.print_first_word(words)
14 All
15 >>> ex25.print_last_word(words)
16 wait.
17 >>> wrods
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20 NameError: name 'wrods' is not defined
21 >>> words
22 ['good', 'things', 'come', 'to', 'those', 'who']
23 >>> ex25.print_first_word(sorted_words)
24 All
25 >>> ex25.print_last_word(sorted_words)
26 who
27 >>> sorted_words
28 ['come', 'good', 'things', 'those', 'to', 'wait.']
29 >>> sorted_words = ex25.sort_sentence(sentence)
30 >>> sorted_words
31 ['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
32 >>> ex25.print_first_and_last(sentence)
33 All
34 wait.
35 >>> ex25.print_first_and_last_sorted(sentence)
36 All
37 who
38 >>> ^D
39 $
```

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 other imports you've done. Notice you don't 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`, you'll learn about that 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 is interesting. I made a mistake and typed the `words` variable as `wrods` so python gave me an error on Lines 18-20.
- Line 21-22 is where 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 extra credit.

Extra Credit

1. Take the remaining lines of the WYSS output and figure out what they're 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 also notice how the help is those odd `"""` strings you put after each function in ex25? Those special strings are called documentation comments and we'll be seeing more of them.
3. Type `ex25.` is annoying, so a shortcut is do your import like this: `from ex25 import *` which is like saying, "Import everything from ex25." Programmers like saying things backwards. Start a new session and see how all your functions are right there now.
4. Try breaking your file and see what it looks like in `python` when you use it. You will have to quit python with CTRL-D (CTRL-Z on windows) to be able to reload it.

Exercise 26: Congratulations, Take A Test!

You are almost done with the first half of the book. The second half will start teaching you logic where things become more interesting and you'll start to be able to do useful things.

Before you continue though, I have a quiz for you. This quiz will be *very* hard because it requires you to fix someone else's code. When you're a programmer you have to deal with other programmer's code, and also with their arrogance. They will very frequently claim that their code is perfect. Despite all of the obvious flaws in what they've written, it couldn't possibly be anything they've written.

These programmers are stupid people who care little for others. A good programmer assumes, like a good scientist, that there's always *some* probability their code is wrong. Good programmers start from the premise that their software is broken and then work to rule out all possible ways it could be wrong before finally admitting that maybe it really is the other guy's code.

In this exercise, you will practice dealing with a bad programmer by fixing a bad programmer's code. I have poorly copied exercises 24 and 25 into a file and then removed random characters and added flaws. Most of the errors are things Python will tell you, some of them are math errors you should find. Others are formatting errors or spelling mistakes in the strings.

All of them are very common errors programmers make. Even experienced ones.

Your job in this exercise is to correct this file. Use all of your skills to make this file better. Analyze it first, maybe printing it out to edit it like you would a school term paper. Fix each flaw and keep running it and fixing it until you can. Try not to get help, and instead if you get stuck take a break and come back to it later.

Even if this takes days to do, bust through it and make it right.

Finally, the point of this exercise isn't to type it in, but to fix an existing file. To do that, you must go to:

- http://learnpythonthehardway.com/wiki/Exercise26

With your web browser, and copy-paste the code into a file to work on name `ex26.py`.

Exercise 27: Memorizing Logic

Today is the day you start learning about logic. Up to this point you have done everything you possibly can reading and writing files, to the terminal, and have learned quite a lot of the math capabilities of Python.

From now on though, you will be learning about logic, but just enough logic to be dangerous. You won't learn complex theories that academics love to study, but instead just the simple basic logic that makes real programs work and that real programmers need every day.

However, learning logic has to come after you do some memorization. I want you to do this exercise for an entire week. Do not falter. Even if you are bored out of your mind, keep doing it. This exercise has a set of logic tables you must memorize to make it easier for you to do the later exercises.

I'm warning you this won't be fun at first. It will be downright boring and tedious but this is to teach you a very important skill you'll need as a programmer. You *will* need to be able to memorize important concepts as you go in your life. Most of these concepts will be exciting once you get them. You'll struggle with them, like wrestling a squid, then one day *snap* you'll understand it. All that work memorizing the basics pays off big later.

Here's a tip on how to memorize something without going insane: Do it a little tiny bit at a time throughout the day and mark down what you need to work on most. Don't try to sit down for 2 hours straight and memorize these tables as that won't work. Your brain will really only retain whatever you studied in the first 15 or 30 minutes anyway.

Instead, what you should do is create a bunch of index cards with each column on the left on one side (True or False) and the column on the right on the back. You should then pull them out, see the "True or False" and be able to immediately say "True!" Keep practicing until you can do this.

Once you can do that, start writing out your own truth tables each night into a notebook. Don't just copy them, but instead try to do them from memory, and when you get stuck glance quickly at the ones I have here to refresh your memory. Doing this will train your brain to remember the whole table.

Don't spend more than one week on this, because you'll be applying it as you go. You might want to keep your index cards as a "warm up" before you sit down to do further exercises though.

The Truth Terms

In python we have the following terms (characters and phrases) for determining if something is "True" or "False". Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

- `and`
- `or`
- `not`
- `!=` (not equal)
- `==` (equal)

- >= (greater-than-equal)
- <= (less-than-equal)
- True
- False

You actually have run into these characters before, but maybe not the phrases. The phrases (and, or, not) actually work the way you expect them to, just like in English.

The Truth Tables

We will now use these characters to make the truth tables you need to memorize.

NOT	True?
not False	True
not True	False

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

NOT AND	True?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

!=	True?
1 != 0	True
1 != 1	False
0 != 1	True
0 != 0	False

==	True?
1 == 0	False
1 == 1	True
0 == 1	False
0 == 0	True

Now use these tables to write up your own cards and spend the week memorizing them.

Advice From An Old Programmer

You've finished this book and now you have decided to continue on with programming. Maybe it will be a career for you, or maybe you'll just do it as a hobby. For whatever reason you'll need some advice to make sure you continue on the right path and get the most enjoyment out of your newly chosen hobby.

I have been programming for a very long time. So long that it is incredibly boring to me. At the time that I wrote this book I knew about 20 programming languages and could learn new ones in about a day to a week depending on how weird they were. Eventually though this just became boring and couldn't hold my interest.

What I discovered after this journey of learning was that the languages didn't matter, it was what you did with them. Actually, I always knew that, but I'd get distracted by the languages and forget it periodically. Now I never forget it, and neither should you.

The programming language you learn and use does not matter. Do *not* get sucked into the religion surrounding programing languages as that will only blind you to their true purpose of being your tool for doing interesting things.

Programming as an intellectual activity is the *only* art form that allows you to create interactive art. You can create projects that other people can play with and you can talk to them indirectly. No other art form is quite this interactive. Movies flow to the audience in one direction. Paintings don't move. Code goes both ways.

Programming as a profession is only moderately interesting. It can be a good job, but if you want to make about the same money and be happier you could actually just go run a fast food joint. You are much better off using code as your secret weapon in another profession.

People who can code in the world of technology companies are a dime a dozen and get no respect. People who can code in biology, medicine, government, sociology, physics, history, and mathematics are respected and can do amazing things to advance those disciplines.

Of course, all of this advice is pointless. If you liked learning to write software with this book then you should try to use it to improve your life anyway you can. You should go out and explore this weird wonderful new intellectual pursuit that barely anyone in the last 50 years has been able to explore. Might as well enjoy it while you can.

Finally, I will say that learning to create software changes you and makes you different. Not better or worse, just different. You may find that people treat you harshly because you can create software, maybe using words like "nerd". Maybe you'll find that because you can dissect their logic that they hate arguing with you. You may even find that simply knowing how a computer works makes you annoying and weird to them.

To this I only have one piece of advice: they can go to hell. The world needs more weird people who know how things work and who love to figure it all out. When they treat you like this, just remember that this is *your* journey, not theirs. Being different is not a crime, and people who tell you it is are just jealous that you've picked up a skill they never in their wildest dreams could acquire.

You can code. They cannot. That is pretty damn cool.

Indices and tables

- *Index*
- *Module Index*
- *Search Page*