Non-Programmer's Tutorial for Python 3/Print version

Contents

1. Front matter

This tutorial is more or less a conversion of Non-Programmer's Tutorial for Python 2.6. Older versions and some versions in Korean, Spanish, Italian and Greek are available from http://jjc.freeshell.org/easytut/

The *Non-Programmers' Tutorial For Python 3* is a tutorial designed to be an introduction to the Python programming language. This guide is for someone with no programming experience.

If you have programmed in other languages I recommend using Python Tutorial for Programmers (https://docs.python.org/3/tutorial/index.html) written by Guido van Rossum.

If you have any questions or comments please use the discussion pages or see Authors page for author contact information. I welcome questions and comments about this tutorial. I will try to answer any questions you have as best I can.

Thanks go to James A. Brown for writing most of the Windows install info. Thanks also to Elizabeth Cogliati for complaining enough :) about the original tutorial (that is almost unusable for a non-programmer), for proofreading, and for many ideas and comments on it. Thanks to Joe Oppegaard for writing almost all the exercises. Thanks to everyone I have missed.

Other resources

- Python Home Page (http://www.python.org)
- Python 3 Documentation (https://docs.python.org/3/)
- A Byte of Python by Swaroop C H (http://www.swaroopch.com/notes/python)
- Porting to Python 3: An in-depth guide (http://python3porting.com/)

---

2. Intro

First things first

So, you've never programmed before. As we go through this tutorial, I will attempt to teach you how to program. There really is only one way to learn to program. **You** must read *code* and write *code* (as computer programs are often called). I'm going to show you lots of code. You should type in code that I show you to see what happens. Play around with it and make changes. The worst that can happen is that it won't work. When I type in code it will be formatted like this:

```
##Python is easy to learn
print("Hello, World!")
```

That's so it is easy to distinguish from the other text. If you're reading this on the Web, you'll notice the code is in color -- that's just to make it stand out, and to make the different parts of the code stand out from each other. The code you enter will probably not be colored, or the colors may be different, but it won't affect the code as long as you enter it the same way as it's printed here.

If the computer prints something out it will be formatted like this:

```
Hello, World!
```

(Note that printed text goes to your screen, and does not involve paper. Before computers had screens, the output of computer programs would be printed on paper.)

Note that this is a Python 3 tutorial, which means that most of the examples will not work in Python 2.7 and before. As well, some of the extra libraries (third-party libraries) have not yet been converted. You may want to consider learning from the Non-Programmer's Tutorial for Python 2.6. However, the differences between versions are not particularly large, so if you learn one, you should be able to read programs written for the other without much difficulty.

There will often be a mixture of the text you type (which is shown in **bold**) and the text the program prints to the screen, which would look like this:

```
Halt!
Who Goes there? Josh
You may pass, Josh
```

(Some of the tutorial has not been converted to this format. Since this is a wiki, you can convert it when you find it.)

I will also introduce you to the terminology of programming - for example, that programming is often referred to as *coding* or *hacking*. This will not only help you understand what programmers are talking about, but also help the learning process.

Now, on to more important things. In order to program in Python you need the Python 3 software. If you don't already have the Python software go to www.python.org/download (http://www.python.org/download/) and get the proper version for your platform. Download it, read the instructions and get it installed.

Installing Python

For Python programming you need a working Python installation and a text editor. Python comes with its own editor, *IDLE*, which is quite nice and totally sufficient for the beginning. As you get more into programming, you will probably switch to some other editor like `emacs`, `vi` or another.

The Python download page is http://www.python.org/download. The most recent version is Python 3.4.3 (as of February 2015); **Python 2.7 and older versions will not work with this tutorial.** There are various different installation files for different computer platforms available on the download site. Here are some specific instructions for the most common operating systems:

Linux, BSD, and Unix users

You are probably lucky and Python is already installed on your machine. To test it type `python3` on a command line. If you see something like what is shown in the following section, you are set.

IDLE may need to be installed separately, from its own package such as `idle3` or as part of `python-tools`.

If you have to install Python, first try to use the operating system's package manager or go to the repository where your packages are available and get Python 3. Python 3.0 was released in December 2008; all distributions should have Python 3 available, so you may not need to compile it from scratch. Ubuntu and Fedora do have Python 3 binary packages available, but they are not yet the default, so they need to be

installed specially.

Roughly, here are the steps to compile Python from source code in Unix (If these totally don't make sense, you may want to read another introduction to *nix, such as Introduction to Linux (http://tldp.org/LDP/intro-linux/html/index.html)):

- Download the .tgz file (use your Web browser to get the gzipped tar file from https://www.python.org/downloads/release/python-343)
- Uncompress the tar file (put in the correct path to where you downloaded it):

```
$ tar -xvzf ~/Download/Python-3.4.3.tgz
... list of files as they are uncompressed
```

- Change to the directory and tell the computer to compile and install the program

```
$ cd Python-3.4/
$ ./configure --prefix=$HOME/python3_install
 ... lots of output.  Watch for error messages here ...
$ make
 ... even more output.  Hopefully no error messages ...
$ make install
```

- Add Python 3 to your path. You can test it first by specifying the full path. You should add $HOME/python3_install/bin to your PATH bash variable.

```
$ ~/python3_install/bin/python3
Python 3.4.3 (... size and date information ...)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The above commands will install Python 3 to your home directory, which is probably what you want, but if you skip the `--prefix=$HOME/python3_install`, it will install it to `/usr/local`. If you want to use the IDLE graphical code editor, you need to make sure that the `tk` and `tcl` libraries, together with their development files, are installed on the system. You will get a warning during the `make` phase if these are not available.

Mac users

Starting from Mac OS X Tiger, Python ships by default with the operating system, but you will need to update to Python 3 until OS X starts including Python 3 (check the version by starting `python3` in a command line terminal). Also IDLE (the Python editor) might be missing in the standard installation. If you want to (re-)install Python, get the MacOS installer from the Python download site (https://www.python.org/downloads/release/python-343/).

Windows users

Download the appropriate Windows installer (the x86 MSI installer (https://www.python.org/ftp/python/3.4.3/python-3.4.3.msi), if you do not have a 64-bit AMD or Intel chip). Start the installer by double-clicking it and follow the prompts.

See https://docs.python.org/3/using/windows.html#installing-python for more information.

Configuring your PATH environment variable

The PATH environment variable is a list of folders, separated by semicolons, in which Windows will look for a program whenever you try to execute one by typing its name at a Command Prompt. You can see the current value of your PATH by typing this command at a Command Prompt:

```
echo %PATH%
```

The easiest way to permanently change environment variables is to bring up the built-in environment variable editor in Windows. How you get to this editor is slightly different on different versions of Windows.

On Windows 8: Press the Windows key and type *Control Panel* to locate the Windows Control Panel. Once you've opened the Control Panel, select View by: Large Icons, then click on *System*. In the window that pops up, click the *Advanced System Settings* link, then click the *Environment Variables...* button.

On Windows 7 or **Vista**: Click the Start button in the lower-left corner of the screen, move your mouse over *Computer*, right-click, and select *Properties* from the pop-up menu. Click the *Advanced System Settings* link, then click the *Environment Variables...* button.

On Windows XP: Right-click the *My Computer* icon on your desktop and select *Properties*. Select the *Advanced* tab, then click the *Environment Variables...* button.

Once you've brought up the environment variable editor, you'll do the same thing regardless of which version of Windows you're running. Under *System Variables* in the bottom half of the editor, find a variable called `PATH`. If there is is one, select it and click *Edit...*. Assuming your Python root is `C:\Python34`, add these two folders to your path (and make sure you get the semicolons right; there should be a semicolon between each folder in the list):

```
C:\Python34
C:\Python34\Scripts
```

Note: If you want to double-click and start your Python programs from a Windows folder and not have the console window disappear, you can add the following code to the bottom of each script:

```
#stops console from exiting
end_prog = ""
while end_prog != "q":
        end_prog = input("type q to quit")
```

Interactive Mode

Go into IDLE (also called the Python GUI). You should see a window that has some text like this:

```
Python 3.0 (r30:67503, Dec 29 2008, 21:31:07)
[GCC 4.3.2 20081105 (Red Hat 4.3.2-7)] on linux2
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 3.0
>>>
```

The >>> is Python's way of telling you that you are in interactive mode. In interactive mode what you type is

immediately run. Try typing `1+1` in. Python will respond with `2`. Interactive mode allows you to test out and see what Python will do. If you ever feel you need to play with new Python statements, go into interactive mode and try them out.

Creating and Running Programs

Go into IDLE if you are not already. In the menu at the top, select *File* then *New File*. In the new window that appears, type the following:

```
print("Hello, World!")
```

Now save the program: select *File* from the menu, then *Save*. Save it as "`hello.py`" (you can save it in any folder you want). Now that it is saved it can be run.

Next run the program by going to *Run* then *Run Module* (or if you have an older version of IDLE use *Edit* then *Run script*). This will output `Hello, World!` on the **Python Shell** window.

For a more in-depth introduction to IDLE, a longer tutorial with screenshots can be found at http://hkn.eecs.berkeley.edu/~dyoo/python/idle_intro/index.html.

Program file names

It is very useful to stick to some rules regarding the file names of Python programs. Otherwise some things *might* go wrong unexpectedly. These don't matter as much for programs, but you can have weird problems if you don't follow them for module names (modules will be discussed later).

1. Always save the program with the extension `.py`. Do not put another dot anywhere else in the file name.
2. Only use standard characters for file names: letters, numbers, dash (`-`) and underscore (_).
3. White space (" ") should not be used at all (use underscores instead).
4. Do not use anything other than a letter (particularly no numbers!) at the beginning of a file name.
5. Do not use "non-English" characters (such as å, ƃ, ç, ð, é, õ, ü) in your file names—or, even better, do not use them at all when programming.

Using Python from the command line

If you don't want to use Python from the command line, you don't have to, just use IDLE. To get into interactive mode just type `python3` without any arguments. To run a program, create it with a text editor (Emacs has a good Python mode) and then run it with `python3` *`program_name`*.

Additionally, to use Python within Vim, you may want to visit the Python wiki page about VIM (http://wiki.python.org/moin/Vim).

Running Python Programs in *nix

If you are using Unix (such as Linux, Mac OS X, or BSD), if you make the program executable with chmod, and have as the first line:

```
#!/usr/bin/env python3
```

you can run the python program with `./hello.py` like any other command.

Where to get help

At some point in your Python career you will probably get stuck and have no clue about how to solve the problem you are supposed to work on. This tutorial only covers the basics of Python programming, but there is a lot of further information available.

Python documentation

First of all, Python is very well documented. There might even be copies of these documents on your computer that came with your Python installation:

- The official Python 3 Tutorial (http://docs.python.org/3/tutorial/) by Guido van Rossum is often a good starting point for general questions.
- For questions about standard modules (you will learn what these are later), the Python 3 Library Reference (http://docs.python.org/3/library/) is the place to look.
- If you really want to get to know something about the details of the language, the Python 3 Reference Manual (http://docs.python.org/3/reference/) is comprehensive but quite complex for beginners.

Python user community

There are a lot of other Python users out there, and usually they are nice and willing to help you. This very active user community is organised mostly through mailing lists and a newsgroup:

- The tutor mailing list (http://mail.python.org/mailman/listinfo/tutor) is for folks who want to ask questions regarding how to learn computer programming with the Python language.
- The python-help mailing list (http://www.python.org/community/lists/#python-help) is python.org's help desk. You can ask a group of knowledgeable volunteers questions about all your Python problems.
- The Python newsgroup comp.lang.python (news:comp.lang.python) (Google groups archive (http://groups.google.com/group/comp.lang.python/)) is the place for general Python discussions, questions and the central meeting point of the community.
- Python wiki has a list of local user groups (http://wiki.python.org/moin/LocalUserGroups), you can join the group mailing list and ask questions. You can also participate in the user group meetings.
- LearnPython (https://www.reddit.com/r/learnpython) subreddit is another location where beginner level questions can be asked.

In order not to reinvent the wheel and discuss the same questions again and again, people will appreciate very much if you *do a web search for a solution to your problem before contacting these lists!*

---

3. Hello, World

What you should know

Once you've read and mastered this chapter, you should know how to edit programs in a text editor or IDLE, save them to the hard disk, and run them once they have been saved.

Printing

Programming tutorials since the beginning of time have started with a little program called "Hello, World!"[1]

So here it is:

```
print("Hello, World!")
```

If you are using the command line to run programs then type it in with a text editor, save it as `hello.py` and run it with `python3 hello.py`

Otherwise go into IDLE, create a new window, and create the program as in section Creating and Running Programs.

When this program is run here's what it prints:

```
Hello, World!
```

Now I'm not going to tell you this every time, but when I show you a program I recommend that you type it in and run it. I learn better when I type it in and you probably do too.

Now here is a more complicated program:

```
print("Jack and Jill went up a hill")
print("to fetch a pail of water;")
print("Jack fell down, and broke his crown,")
print("and Jill came tumbling after.")
```

When you run this program it prints out:

```
Jack and Jill went up a hill
to fetch a pail of water;
Jack fell down, and broke his crown,
and Jill came tumbling after.
```

When the computer runs this program it first sees the line:

```
print("Jack and Jill went up a hill")
```

so the computer prints:

```
Jack and Jill went up a hill
```

Then the computer goes down to the next line and sees:

```
print("to fetch a pail of water;")
```

So the computer prints to the screen:

```
to fetch a pail of water;
```

The computer keeps looking at each line, follows the command and then goes on to the next line. The computer keeps running commands until it reaches the end of the program.

Terminology

Now is probably a good time to give you a bit of an explanation of what is happening - and a little bit of programming terminology.

What we were doing above was using a *function* called `print`. The function's name - `print` - is followed by parentheses containing zero or more *arguments*. So in this example

```
print("Hello, World!")
```

there is one *argument*, which is `"Hello, World!"`. Note that this argument is a group of characters enclosed in double quotes (""). This is commonly referred to as a *string of characters*, or *string*, for short. Another example of a string is `"Jack and Jill went up a hill"`. The combination of a function and parentheses with the arguments is a *function call*.

A function and its arguments are one type of *statement* that python has, so

```
print("Hello, World!")
```

is an example of a statement. Basically, you can think of a statement as a single line in a program.

That's probably more than enough terminology for now.

Expressions

Here is another program:

```
print("2 + 2 is", 2 + 2)
print("3 * 4 is", 3 * 4)
print("100 - 1 is", 100 - 1)
print("(33 + 2) / 5 + 11.5 is", (33 + 2) / 5 + 11.5)
```

And here is the *output* when the program is run:

```
2 + 2 is 4
3 * 4 is 12
100 - 1 is 99
(33 + 2) / 5 + 11.5 is 18.5
```

As you can see, Python can turn your thousand-dollar computer into a five-dollar calculator.

Arithmetic expressions

In this example, the print function is followed by two arguments, with each of the arguments separated by a comma. So with the first line of the program

```
print("2 + 2 is", 2 + 2)
```

The first argument is the string `"2 + 2 is"` and the second argument is the *arithmetic expression* `2 + 2`, which is one kind of *expression*.

What is important to note is that a string is printed as is (without the enclosing double quotes), but an *expression* is *evaluated*, or converted to its actual value.

Python has seven basic operations for numbers:

Operation	Symbol	Example
Power (exponentiation)	`**`	`5 ** 2 == 25`
Multiplication	`*`	`2 * 3 == 6`
Division	`/`	`14 / 3 == 4.666666666666667`
Integer Division	`//`	`14 // 3 == 4`
Remainder (modulo)	`%`	`14 % 3 == 2`
Addition	`+`	`1 + 2 == 3`
Subtraction	`-`	`4 - 3 == 1`

Notice that there are two ways to do division, one that returns the repeating decimal, and the other that can get the remainder and the whole number. The order of operations is the same as in math:

- parentheses `()`
- exponents `**`
- multiplication `*`, division `/`, integer division `//`, and remainder `%`
- addition `+` and subtraction `-`

So use parentheses to structure your formulas when needed.

Talking to humans (and other intelligent beings)

Often in programming you are doing something complicated and may not in the future remember what you did. When this happens the program should probably be commented. A *comment* is a note to you and other programmers explaining what is happening. For example:

```
# Not quite PI, but a credible simulation
print(22 / 7)
```

Which outputs

```
3.14285714286
```

Notice that the comment starts with a hash: `#`. Comments are used to communicate with others who read the program and your future self to make clear what is complicated.

Note that any text can follow a comment, and that when the program is run, the text after the `#` through to the end of that line is ignored. The `#` does not have to be at the beginning of a new line:

```
# Output PI on the screen
print(22 / 7) # Well, just a good approximation
```

Examples

Each chapter (eventually) will contain examples of the programming features introduced in the chapter. You should at least look over them and see if you understand them. If you don't, you may want to type them in and see what happens. Mess around with them, change them and see what happens.

Denmark.py

```
print("Something's rotten in the state of Denmark.")
print("                -- Shakespeare")
```

Output:

```
Something's rotten in the state of Denmark.
                -- Shakespeare
```

School.py

```
# This is not quite true outside of USA
# and is based on my dim memories of my younger years
print("Firstish Grade")
print("1 + 1 =", 1 + 1)
print("2 + 4 =", 2 + 4)
print("5 - 2 =", 5 - 2)
print()
print("Thirdish Grade")
print("243 - 23 =", 243 - 23)
print("12 * 4 =", 12 * 4)
print("12 / 3 =", 12 / 3)
print("13 / 3 =", 13 // 3, "R", 13 % 3)
print()
print("Junior High")
print("123.56 - 62.12 =", 123.56 - 62.12)
print("(4 + 3) * 2 =", (4 + 3) * 2)
print("4 + 3 * 2 =", 4 + 3 * 2)
print("3 ** 2 =", 3 ** 2)
```

Output:

```
Firstish Grade
1 + 1 = 2
2 + 4 = 6
5 - 2 = 3

Thirdish Grade
243 - 23 = 220
12 * 4 = 48
12 / 3 = 4
13 / 3 = 4 R 1

Junior High
123.56 - 62.12 = 61.44
(4 + 3) * 2 = 14
4 + 3 * 2 = 10
3 ** 2 = 9
```

Exercises

1. Write a program that prints your full name and your birthday as separate strings.
2. Write a program that shows the use of all 7 math functions.

Solution

1. Write a program that prints your full name and your birthday as separate strings.

```
print("Ada Lovelace", "born on", "November 27, 1852")
```

```
print("Albert Einstein", "born on", "14 March 1879")
```

```
print(("John Smith"), ("born on"), ("14 March 1879"))
```

Solution

2. Write a program that shows the use of all 7 arithmetic operations.

```
print("5**5 = ", 5**5)
print("6*7 = ", 6*7)
print("56/8 = ", 56/8)
print("14//6 = ", 14//6)
print("14%6 = ", 14%6)
print("5+6 = ", 5+6)
print("9-0 = ", 9-0)
```

Footnotes

1. Here is a great list of the famous "Hello, world!" program in many programming languages. Just so you know how simple Python can be...

---

4. Who Goes There?

Input and Variables

Now I feel it is time for a really complicated program. Here it is:

```
print("Halt!")
user_input = input("Who goes there? ")
print("You may pass,",  user_input)
```

When **I** ran it, here is what **my** screen showed:

```
Halt!
Who goes there? Josh
You may pass, Josh
```

Note: After running the code by pressing F5, the python shell will only give output:

```
Halt!
Who goes there?
```

You need to enter your name in the python shell, and then press enter for the rest of the output.

Of course when you run the program your screen will look different because of the `input()` statement. When you ran the program you probably noticed (you did run the program, right?) how you had to type in

your name and then press Enter. Then the program printed out some more text and also your name. This is an example of *input*. The program reaches a certain point and then waits for the user to input some data that the program can use later.

Of course, getting information from the user would be useless if we didn't have anywhere to put that information and this is where variables come in. In the previous program `user_input` is a *variable*. Variables are like a box that can store some piece of data. Here is a program to show examples of variables:

```
a = 123.4
b23 = 'Spam'
first_name = "Bill"
b = 432
c = a + b
print("a + b is",c)
print("first_name is",first_name)
print("Sorted Parts, After Midnight or",b23)
```

And here is the output:

```
a + b is 555.4
first_name is Bill
Sorted Parts, After Midnight or Spam
```

Variables store data. The variables in the above program are `a`, `b23`, `first_name`, `b`, and `c`. The two basic types are *strings* and *numbers*. Strings are a sequence of letters, numbers and other characters. In this example `b23` and `first_name` are variables that are storing strings. `Spam`, `Bill`, `a + b is`, `first_name is`, and `Sorted Parts, After Midnight or` are the strings in this program. The characters are surrounded by `"` or `'`. The other type of variables are numbers. Remember that variables are used to store a value, they do not use quotation marks (" and '). If you want to use an actual *value*, you *must* use quotation marks.

```
value1 == Pim
value2 == "Pim"
```

Both look the same, but in the first one Python checks if the value stored in the variable `value1` is the same as the value stored in the *variable* `Pim`. In the second one, Python checks if the string (the actual letters `P`,`i`, and `m`) are the same as in `value2` (continue this tutorial for more explanation about strings and about the `==`).

Assignment

Okay, so we have these boxes called variables and also data that can go into the variable. The computer will see a line like `first_name = "Bill"` and it reads it as "Put the string `Bill` into the box (or variable) `first_name`". Later on it sees the statement `c = a + b` and it reads it as "put the sum of `a + b` or `123.4 + 432` which equals `555.4` into `c`". The right hand side of the statement (`a + b`) is *evaluated* and the result is stored in the variable on the left hand side (`c`). This is called *assignment*, and you should not confuse the assignment equal sign (=) with "equality" in a mathematical sense here (that's what == will be used for later).

Here is another example of variable usage:

```
a = 1
print(a)
a = a + 1
print(a)
a = a * 2
print(a)
```

And of course here is the output:

```
1
2
4
```

Even if the same variable appears on both sides of the equals sign (e.g., spam = spam), the computer still reads it as, "First find out the data to store and then find out where the data goes."

One more program before I end this chapter:

```
number = float(input("Type in a number: "))
integer = int(input("Type in an integer: "))
text = input("Type in a string: ")
print("number =", number)
print("number is a", type(number))
print("number * 2 =", number * 2)
print("integer =", integer)
print("integer is a", type(integer))
print("integer * 2 =", integer * 2)
print("text =", text)
print("text is a", type(text))
print("text * 2 =", text * 2)
```

The output I got was:

```
Type in a number: 12.34
Type in an integer: -3
Type in a string: Hello
number = 12.34
number is a <class 'float'>
number * 2 = 24.68
integer = -3
integer is a <class 'int'>
integer * 2 = -6
text = Hello
text is a <class 'str'>
text * 2 = HelloHello
```

Notice that `number` was created with `float(input())` while `text` was created with `input()`. `input()` returns a string while the function `float` returns a number from a string. `int` returns an integer, that is a number with no decimal point. When you want the user to type in a decimal use `float(input())`, if you want the user to type in an integer use `int(input())`, but if you want the user to type in a string use `input()`.

The second half of the program uses the `type()` function which tells what kind a variable is. Numbers are of type `int` or `float`, which are short for *integer* and *floating point* (mostly used for decimal numbers), respectively. Text strings are of type `str`, short for *string*. Integers and floats can be worked on by mathematical functions, strings cannot. Notice how when python multiplies a number by an integer the expected thing happens. However when a string is multiplied by an integer the result is that multiple copies of the string are produced (i.e., `text * 2 = HelloHello`).

Operations with strings do different things than operations with numbers. As well, some operations only work with numbers (both integers and floating point numbers) and will give an error if a string is used. Here are some interactive mode examples to show that some more.

```
>>> print("This" + " " + "is" + " joined.")
This is joined.
>>> print("Ha, " * 5)
```

```
Ha, Ha, Ha, Ha, Ha,
>>> print("Ha, " * 5 + "ha!")
Ha, Ha, Ha, Ha, Ha, ha!
>>> print(3 - 1)
2
>>> print("3" - "1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
>>>
```

Here is the list of some string operations:

Operation	Symbol	Example
Repetition	*	"i" * 5 == "iiiii"
Concatenation	+	"Hello, " + "World!" == "Hello, World!"

Examples

Rate_times.py

```
# This program calculates rate and distance problems
print("Input a rate and a distance")
rate = float(input("Rate: "))
distance = float(input("Distance: "))
print("Time:", (distance / rate))
```

Sample runs:

```
Input a rate and a distance
Rate: 5
Distance: 10
Time: 2.0
```

```
Input a rate and a distance
Rate: 3.52
Distance: 45.6
Time: 12.9545454545
```

Area.py

```
# This program calculates the perimeter and area of a rectangle
print("Calculate information about a rectangle")
length = float(input("Length: "))
width = float(input("Width: "))
print("Area:", length * width)
print("Perimeter:", 2 * length + 2 * width)
```

Sample runs:

```
Calculate information about a rectangle
Length: 4
Width: 3
Area: 12.0
Perimeter: 14.0
```

```
Calculate information about a rectangle
Length: 2.53
```

```
Width: 5.2
Area: 13.156
Perimeter: 15.46
```

Temperature.py

```
# This program converts Fahrenheit to Celsius
fahr_temp = float(input("Fahrenheit temperature: "))
print("Celsius temperature:", (fahr_temp - 32.0) * 5.0 / 9.0)
```

Sample runs:

```
Fahrenheit temperature: 32
Celsius temperature: 0.0
```

```
Fahrenheit temperature: -40
Celsius temperature: -40.0
```

```
Fahrenheit temperature: 212
Celsius temperature: 100.0
```

```
Fahrenheit temperature: 98.6
Celsius temperature: 37.0
```

Exercises

Write a program that gets 2 string variables and 2 number variables from the user, concatenates (joins them together with no spaces) and displays the strings, then multiplies the two numbers on a new line.

Solution

Write a program that gets 2 string variables and 2 number variables from the user, concatenates (joins them together with no spaces) and displays the strings, then multiplies the two numbers on a new line.

```
string1 = input('String 1: ')
string2 = input('String 2: ')
float1 = float(input('Number 1: '))
float2 = float(input('Number 2: '))
print(string1 + string2)
print(float1 * float2)
```

---

5. Count to 10

While loops

Presenting our first *control structure*. Ordinarily the computer starts with the first line and then goes down from there. Control structures change the order that statements are executed or decide if a certain statement will be run. Here's the source for a program that uses the while control structure:

```
a = 0            # FIRST, set the initial value of the variable a to 0(zero).
while a < 10:    # While the value of the variable a is less than 10 do the following:
    a = a + 1    # Increase the value of the variable a by 1, as in: a = a + 1!
    print(a)     # Print to screen what the present value of the variable a is.
                 # REPEAT! until the value of the variable a is equal to 9!? See note.

                 # NOTE:
                 # The value of the variable a will increase by 1
                 # with each repeat, or loop of the 'while statement BLOCK'.
                 # e.g. a = 1 then a = 2 then a = 3 etc. until a = 9 then...
                 # the code will finish adding 1 to a (now a = 10), printing the
                 # result, and then exiting the 'while statement BLOCK'.
                 #              --
                 # While a < 10: |
                 #     a = a + 1 |<--[ The while statement BLOCK ]
                 #     print (a) |
                 #               --
```

And here is the extremely exciting output:

```
1
2
3
4
5
6
7
8
9
10
```

(And you thought it couldn't get any worse after turning your computer into a five-dollar calculator?)

So what does the program do? First it sees the line `a = 0` and sets a to zero. Then it sees `while a < 10:` and so the computer checks to see if `a < 10`. The first time the computer sees this statement, a is zero, so it is less than 10. In other words, as long as a is less than ten, the computer will run the tabbed in statements. This eventually makes a equal to ten (by adding one to a again and again) and the `while a < 10` is not true any longer. Reaching that point, the program will stop running the indented lines.

Always remember to put a colon "**:**" at the end of the `while` statement line!

Here is another example of the use of `while`:

```
a = 1
s = 0
print('Enter Numbers to add to the sum.')
print('Enter 0 to quit.')
while a != 0:
    print('Current Sum:', s)
    a = float(input('Number? '))
    s = s + a
print('Total Sum =', s)
```

```
Enter Numbers to add to the sum.
Enter 0 to quit.
Current Sum: 0
Number? 200
Current Sum: 200.0
Number? -15.25
Current Sum: 184.75
Number? -151.85
Current Sum: 32.9
Number? 10.00
Current Sum: 42.9
Number? 0
Total Sum = 42.9
```

Notice how `print('Total Sum =', s)` is only run at the end. The `while` statement only affects the lines that are indented with whitespace. The `!=` means does not equal so `while a != 0:` means as long as `a` is not zero run the tabbed statements that follow.

Note that `a` is a floating point number, and not all floating point numbers can be accurately represented, so using `!=` on them can sometimes not work. Try typing in 1.1 in interactive mode.

Infinite loops or Never Ending Loop

Now that we have while loops, it is possible to have programs that run forever. An easy way to do this is to write a program like this:

```
while 1 == 1:
   print("Help, I'm stuck in a loop.")
```

The "==" operator is used to test equality of the expressions on the two sides of the operator, just as "<" was used for "less than" before (you will get a complete list of all comparison operators in the next chapter).

This program will output `Help, I'm stuck in a loop.` until the heat death of the universe or you stop it, because 1 will forever be equal to 1. The way to stop it is to hit the Control (or *Ctrl*) button and *C* (the letter) at the same time. This will kill the program. (Note: sometimes you will have to hit enter after the Control-C.) On some systems, nothing will stop it, short of killing the process--so avoid!

Examples

Fibonacci sequence

Fibonacci-method1.py

```
# This program calculates the Fibonacci sequence
a = 0
b = 1
count = 0
max_count = 20

while count < max_count:
    count = count + 1
    print(a, end=" ")  # Notice the magic end=" " in the print function arguments
                       # that keeps it from creating a new line.
    old_a = a    # we need to keep track of a since we change it.
    a = b
    b = old_a + b
print()  # gets a new (empty) line.
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Note that the output is on a single line because of the extra argument `end=" "` in the `print` arguments.

Fibonacci-method2.py

```
# Simplified and faster method to calculate the Fibonacci sequence
a = 0
b = 1
```

```
count = 0
max_count = 10

while count < max_count:
    count = count + 1
    print(a, b, end=" ")  # Notice the magic end=" "
    a = a + b
    b = a + b
print()  # gets a new (empty) line.
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Fibonacci-method3.py

```
a = 0
b = 1
count = 0
maxcount = 20

#once loop is started we stay in it
while count < maxcount:
    count += 1
    olda = a
    a = a + b
    b = olda
    print(olda,end=" ")
print()
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
```

Enter password

Password.py

```
# Waits until a password has been entered.  Use Control-C to break out without
# the password

#Note that this must not be the password so that the
# while loop runs at least once.
password = str()

# note that != means not equal
while password != "unicorn":
    password = input("Password: ")
print("Welcome in")
```

Sample run:

```
Password: auo
Password: y22
Password: password
Password: open sesame
Password: unicorn
Welcome in
```

Exercises

Write a program that asks the user for a Login Name and password. Then when they type "lock", they need to type in their name and password to unlock the program.

Solution

Write a program that asks the user for a Login Name and password. Then when they type "lock", they need to type in their name and password to unlock the program.

```
name = input("What is your UserName: ")
password = input("What is your Password: ")
print("To lock your computer type lock.")
command = None
input1 = None
input2 = None
while command != "lock":
    command = input("What is your command: ")
while input1 != name:
    input1 = input("What is your username: ")
while input2 != password:
    input2 = input("What is your password: ")
print("Welcome back to your system!")
```

If you would like the program to run continuously, just add a `while 1 == 1:` loop around the whole thing. You will have to indent the rest of the program when you add this at the top of the code, but don't worry, you don't have to do it manually for each line! Just highlight everything you want to indent and click on "Indent" under "Format" in the top bar of the python window.

Another way of doing this could be:

```
name = input('Set name: ')
password = input('Set password: ')
while 1 == 1:
    nameguess=""
    passwordguess=""
    key=""
    while (nameguess != name) or (passwordguess != password):
        nameguess = input('Name? ')
        passwordguess = input('Password? ')
    print("Welcome,", name, ". Type lock to lock.")
    while key != "lock":
        key = input("")
```

Notice the `or` in while `(nameguess != name) or (passwordguess != password)`, which we haven't yet introduced. You can probably figure out how it works.

---

6. Decisions

If statement

As always I believe I should start each chapter with a warm-up typing exercise, so here is a short program to compute the absolute value of an integer:

```
n = int(input("Number? "))
if n < 0:
   print("The absolute value of", n, "is", -n)
else:
```

```
    print("The absolute value of", n, "is", n)
```

Here is the output from the two times that I ran this program:

```
Number? -34
The absolute value of -34 is 34
```

```
Number? 1
The absolute value of 1 is 1
```

So what does the computer do when it sees this piece of code? First it prompts the user for a number with the statement "`n = int(input("Number? "))`". Next it reads the line "`if n < 0:`". If `n` is less than zero Python runs the line "`print("The absolute value of", n, "is", -n)`". Otherwise it runs the line "`print("The absolute value of", n, "is", n)`".

More formally Python looks at whether the *expression* `n < 0` is true or false. An `if` statement is followed by an indented *block* of statements that are run when the expression is true. Optionally after the `if` statement is an `else` statement and another indented *block* of statements. This second block of statements is run if the expression is false.

There are a number of different tests that an expression can have. Here is a table of all of them:

operator	function
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

Another feature of the `if` command is the `elif` statement. It stands for else if and means if the original `if` statement is false but the `elif` part is true, then do the `elif` part. And if neither the `if` or `elif` expressions are true, then do what's in the `else` block. Here's an example:

```
a = 0
while a < 10:
    a = a + 1
    if a > 5:
        print(a, ">", 5)
    elif a <= 3:
        print(a, "<=", 3)
    else:
        print("Neither test was true")
```

and the output:

```
1 <= 3
2 <= 3
3 <= 3
Neither test was true
Neither test was true
6 > 5
7 > 5
```

```
8 > 5
9 > 5
10 > 5
```

Notice how the `elif a <= 3` is only tested when the `if` statement fails to be true. There can be more than one `elif` expression, allowing multiple tests to be done in a single `if` statement.

Examples

```
# This Program Demonstrates the use of the == operator
# using numbers
print(5 == 6)
# Using variables
x = 5
y = 8
print(x == y)
```

And the output

```
False
False
```

high_low.py

```
# Plays the guessing game higher or lower

# This should actually be something that is semi random like the
# last digits of the time or something else, but that will have to
# wait till a later chapter.  (Extra Credit, modify it to be random
# after the Modules chapter)
number = 7
guess = -1

print("Guess the number!")
while guess != number:
    guess = int(input("Is it... "))

    if guess == number:
        print("Hooray! You guessed it right!")
    elif guess < number:
        print("It's bigger...")
    elif guess > number:
        print("It's not so big.")
```

Sample run:

```
Guess the number!
Is it... 2
It's bigger...
Is it... 5
It's bigger...
Is it... 10
It's not so big.
Is it... 7
Hooray! You guessed it right!
```

even.py

```
# Asks for a number.
# Prints if it is even or odd

number = float(input("Tell me a number: "))
if number % 2 == 0:
```

```
    print(int(number), "is even.")
elif number % 2 == 1:
    print(int(number), "is odd.")
else:
    print(number, "is very strange.")
```

Sample runs:

```
Tell me a number: 3
3 is odd.
```

```
Tell me a number: 2
2 is even.
```

```
Tell me a number: 3.4895
3.4895 is very strange.
```

average1.py

```
# keeps asking for numbers until 0 is entered.
# Prints the average value.

count = 0
sum = 0.0
number = 1 # set to something that will not exit the while loop immediately.

print("Enter 0 to exit the loop")

while number != 0:
    number = float(input("Enter a number: "))
    if number != 0:
        count = count + 1
        sum = sum + number
    if number == 0:
        print("The average was:", sum / count)
```

Sample runs

Sample runs:

```
Enter 0 to exit the loop
Enter a number: 3
Enter a number: 5
Enter a number: 0
The average was: 4.0
```

```
Enter 0 to exit the loop
Enter a number: 1
Enter a number: 4
Enter a number: 3
Enter a number: 0
The average was: 2.66666666667
```

average2.py

```
# keeps asking for numbers until count numbers have been entered.
# Prints the average value.

#Notice that we use an integer to keep track of how many numbers,
# but floating point numbers for the input of each number
sum = 0.0
```

```
print("This program will take several numbers then average them")
count = int(input("How many numbers would you like to average: "))
current_count = 0

while current_count < count:
    current_count = current_count + 1
    print("Number", current_count)
    number = float(input("Enter a number: "))
    sum = sum + number

print("The average was:", sum / count)
```

Sample runs:

```
This program will take several numbers then average them
How many numbers would you like to average: 2
Number 1
Enter a number: 3
Number 2
Enter a number: 5
The average was: 4.0
```

```
This program will take several numbers then average them
How many numbers would you like to average: 3
Number 1
Enter a number: 1
Number 2
Enter a number: 4
Number 3
Enter a number: 3
The average was: 2.66666666667
```

Exercises

Write a program that asks the user their name, if they enter your name say "That is a nice name", if they enter "John Cleese" or "Michael Palin", tell them how you feel about them ;), otherwise tell them "You have a nice name."

Solution

```
name = input('Your name: ')
if name == 'Bryn':
    print('That is a nice name.')
elif name == 'John Cleese':
    print('... some funny text.')
elif name == 'Michael Palin':
    print('... some funny text.')
else:
    print('You have a nice name.')
```

Modify the higher or lower program from this section to keep track of how many times the user has entered the wrong number. If it is more than 3 times, print "That must have been complicated." at the end, otherwise print "Good job!"

Solution

```
number = 7
guess = -1
count = 0

print("Guess the number!")
while guess != number:
    guess = int(input("Is it... "))
```

```
    count = count + 1
    if guess == number:
        print("Hooray! You guessed it right!")
    elif guess < number:
        print("It's bigger...")
    elif guess > number:
        print("It's not so big.")

if count > 3:
    print("That must have been complicated.")
else:
    print("Good job!")
```

Write a program that asks for two numbers. If the sum of the numbers is greater than 100, print "That is a big number."

Solution

```
number1 = float(input('1st number: '))
number2 = float(input('2nd number: '))
if number1 + number2 > 100:
    print('That is a big number.')
```

---

7. Debugging

What is debugging?

> "As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs." — *Maurice Wilkes discovers debugging*, 1949

By now if you have been messing around with the programs you have probably found that sometimes the program does something you didn't want it to do. This is fairly common. Debugging is the process of figuring out what the computer is doing and then getting it to do what you want it to do. This can be tricky. I once spent nearly a week tracking down and fixing a bug that was caused by someone putting an `x` where a `y` should have been.

This chapter will be more abstract than previous chapters.

What should the program do?

The first thing to do (this sounds obvious) is to figure out what the program should be doing if it is running correctly. Come up with some test cases and see what happens. For example, let's say I have a program to compute the perimeter of a rectangle (the sum of the length of all the edges). I have the following test cases:

height	width	perimeter
3	4	14
2	3	10
4	4	16
2	2	8
5	1	12

I now run my program on all of the test cases and see if the program does what I expect it to do. If it doesn't then I need to find out what the computer is doing.

More commonly some of the test cases will work and some will not. If that is the case you should try and figure out what the working ones have in common. For example here is the output for a perimeter program (you get to see the code in a minute):

```
Height: 3
Width: 4
perimeter = 15
```

```
Height: 2
Width: 3
perimeter = 11
```

```
Height: 4
Width: 4
perimeter = 16
```

```
Height: 2
Width: 2
perimeter = 8
```

```
Height: 5
Width: 1
perimeter = 8
```

Notice that it didn't work for the first two inputs, it worked for the next two and it didn't work on the last one. Try and figure out what is in common with the working ones. Once you have some idea what the problem is finding the cause is easier. With your own programs you should try more test cases if you need them.

What does the program do?

The next thing to do is to look at the source code. One of the most important things to do while programming is reading source code. The primary way to do this is code walkthroughs.

A code walkthrough starts at the first line, and works its way down until the program is done. `while` loops and `if` statements mean that some lines may never be run and some lines are run many times. At each line you figure out what Python has done.

Lets start with the simple perimeter program. Don't type it in, you are going to read it, not run it. The source code is:

```
height = int(input("Height: "))
```

```
width = int(input("Width: "))
print("perimeter =", width + height + width + width)
```

Question: **What is the first line Python runs?**

Answer: The first line is always run first. In this case it is: `height = int(input("Height: "))`

What does that line do?

Prints `Height:` , waits for the user to type a string in, and then converts the string to an integer variable height.

What is the next line that runs?

In general, it is the next line down which is: `width = int(input("Width: "))`

What does that line do?

Prints `Width:` , waits for the user to type a number in, and puts what the user types in the variable width.

What is the next line that runs?

When the next line is not indented more or less than the current line, it is the line right afterwards, so it is: `print("perimeter = ", width + height + width + width)` (It may also run a function in the current line, but that's a future chapter.)

What does that line do?

First it prints `perimeter =` , then it prints the sum of the values contained within the variables, `width` and `height`, from `width + height + width + width`.

Does `width + height + width + width` calculate the perimeter properly?

Let's see, perimeter of a rectangle is the bottom (width) plus the left side (height) plus the top (width) plus the right side (huh?). The last item should be the right side's length, or the height.

Do you understand why some of the times the perimeter was calculated "correctly"?

It was calculated correctly when the width and the height were equal.

The next program we will do a code walkthrough for is a program that is supposed to print out 5 dots on the screen. However, this is what the program is outputting:

```
. . . .
```

And here is the program:

```
number = 5
while number > 1:
    print(".",end=" ")
    number = number - 1
print()
```

This program will be more complex to walkthrough since it now has indented portions (or control structures). Let us begin.

What is the first line to be run?

The first line of the file: `number = 5`

What does it do?

Puts the number 5 in the variable number.

What is the next line?

The next line is: `while number > 1:`

What does it do?

Well, `while` statements in general look at their expression, and if it is true they do the next indented block of code, otherwise they skip the next indented block of code.

So what does it do right now?

If `number > 1` is true then the next two lines will be run.

So is `number > 1`?

The last value put into `number` was `5` and `5 > 1` so yes.

So what is the next line?

Since the `while` was true the next line is: `print(".",end=" ")`

What does that line do?

Prints one dot and since the extra argument `end=" "` exists the next printed text will not be on a different screen line.

What is the next line?

`number = number - 1` since that is following line and there are no indent changes.

What does it do?

It calculates `number - 1`, which is the current value of `number` (or 5) subtracts 1 from it, and makes that the new value of number. So basically it changes `number`'s value from 5 to 4.

What is the next line?

Well, the indent level decreases so we have to look at what type of control structure it is. It is a `while` loop, so we have to go back to the `while` clause which is `while number > 1:`

What does it do?

It looks at the value of number, which is 4, and compares it to 1 and since `4 > 1` the while loop continues.

What is the next line?

Since the while loop was true, the next line is: `print(".",end=" ")`

What does it do?

It prints a second dot on the line, ending by a space.

What is the next line?

No indent change so it is: `number = number - 1`

And what does it do?

It takes the current value of number (4), subtracts 1 from it, which gives it 3 and then finally makes 3 the new value of number.

What is the next line?

Since there is an indent change caused by the end of the while loop, the next line is: `while number > 1:`

What does it do?

It compares the current value of number (3) to 1. `3 > 1` so the while loop continues.

What is the next line?

Since the while loop condition was true the next line is: `print(".",end=" ")`

And it does what?

A third dot is printed on the line.

What is the next line?

It is: `number = number - 1`

What does it do?

It takes the current value of number (3) subtracts from it 1 and makes the 2 the new value of number.

What is the next line?

Back up to the start of the while loop: `while number > 1:`

What does it do?

It compares the current value of number (2) to 1. Since `2 > 1` the while loop continues.

What is the next line?

Since the while loop is continuing: `print(".",end=" ")`

What does it do?

It discovers the meaning of life, the universe and everything. I'm joking. (I had to make sure you were awake.) The line prints a fourth dot on the screen.

What is the next line?

It's: `number = number - 1`

What does it do?

Takes the current value of number (2) subtracts 1 and makes 1 the new value of number.

What is the next line?

Back up to the while loop: `while number > 1:`

What does the line do?

It compares the current value of number (1) to 1. Since `1 > 1` is false (one is not greater than one), the while loop exits.

What is the next line?

Since the while loop condition was false the next line is the line after the while loop exits, or: `print()`

What does that line do?

Makes the screen go to the next line.

Why doesn't the program print 5 dots?

The loop exits 1 dot too soon.

How can we fix that?

Make the loop exit 1 dot later.

And how do we do that?

There are several ways. One way would be to change the while loop to: `while number > 0:` Another way would be to change the conditional to: `number >= 1` There are a couple others.

How do I fix my program?

You need to figure out what the program is doing. You need to figure out what the program should do. Figure out what the difference between the two is. Debugging is a skill that has to be practiced to be learned. If you can't figure it out after an hour, take a break, talk to someone about the problem or contemplate the lint in your navel. Come back in a while and you will probably have new ideas about the problem. Good luck.

---

8. Defining Functions

Creating Functions

To start off this chapter I am going to give you an example of what you could do but shouldn't (so don't type it in):

```
a = 23
b = -23

if a < 0:
    a = -a
if b < 0:
    b = -b
if a == b:
    print("The absolute values of", a, "and", b, "are equal.")
else:
    print("The absolute values of", a, "and", b, "are different.")
```

with the output being:

```
The absolute values of 23 and 23 are equal.
```

The program seems a little repetitive. Programmers hate to repeat things -- that's what computers are for, after all! (Note also that finding the absolute value changed the value of the variable, which is why it is

printing out 23, and not -23 in the output.) Fortunately Python allows you to create functions to remove duplication. Here is the rewritten example:

```
a = 23
b = -23

def absolute_value(n):
    if n < 0:
        n = -n
    return n

if absolute_value(a) == absolute_value(b):
    print("The absolute values of", a, "and", b, "are equal.")
else:
    print("The absolute values of", a, "and", b, "are different.")
```

with the output being:

```
The absolute values of 23 and -23 are equal.
```

The key feature of this program is the `def` statement. `def` (short for define) starts a function definition. `def` is followed by the name of the function `absolute_value`. Next comes a '(' followed by the parameter `n` (`n` is passed from the program into the function when the function is called). The statements after the ':' are executed when the function is used. The statements continue until either the indented statements end or a `return` is encountered. The `return` statement returns a value back to the place where the function was called. We already have encountered a function in our very first program, the `print` function. Now we can make new functions.

Notice how the values of `a` and `b` are not changed. Functions can be used to repeat tasks that don't return values. Here are some examples:

```
def hello():
    print("Hello")

def area(width, height):
    return width * height

def print_welcome(name):
    print("Welcome", name)

hello()
hello()

print_welcome("Fred")
w = 4
h = 5
print("width =", w, " height =", h, " area =", area(w, h))
```

with output being:

```
Hello
Hello
Welcome Fred
width = 4  height = 5  area = 20
```

That example shows some more stuff that you can do with functions. Notice that you can use no arguments or two or more. Notice also when a function doesn't need to send back a value, a return is optional.

Variables in functions

When eliminating repeated code, you often have variables in the repeated code. In Python, these are dealt with in a special way. So far all variables we have seen are global variables. Functions have a special type of variable called local variables. These variables only exist while the function is running. When a local variable has the same name as another variable (such as a global variable), the local variable hides the other. Sound confusing? Well, these next examples (which are a bit contrived) should help clear things up.

```
a = 4

def print_func():
    a = 17
    print("in print_func a =", a)

print_func()
print("a = ", a)
```

When run, we will receive an output of:

```
in print_func a = 17
a = 4
```

Variable assignments inside a function do not override global variables, they exist only inside the function. Even though a was assigned a new value inside the function, this newly assigned value was only relevant to `print_func`, when the function finishes running, and the a's values is printed again, we see the originally assigned values.

Here is another more complex example.

```
a_var = 10
b_var = 15
e_var = 25

def a_func(a_var):
    print("in a_func a_var =", a_var)
    b_var = 100 + a_var
    d_var = 2 * a_var
    print("in a_func b_var =", b_var)
    print("in a_func d_var =", d_var)
    print("in a_func e_var =", e_var)
    return b_var + 10

c_var = a_func(b_var)

print("a_var =", a_var)
print("b_var =", b_var)
print("c_var =", c_var)
print("d_var =", d_var)
```

output:

```
in a_func a_var =  15
in a_func b_var =  115
in a_func d_var =  30
in a_func e_var =  25
a_var =  10
b_var =  15
c_var =  125
d_var =

Traceback (most recent call last):
 File "C:\def2.py", line 19, in <module>
   print("d_var = ", d_var)
NameError: name 'd_var' is not defined
```

In this example the variables `a_var`, `b_var`, and `d_var` are all local variables when they are inside the function `a_func`. After the statement `return b_var + 10` is run, they all cease to exist. The variable `a_var` is automatically a local variable since it is a parameter name. The variables `b_var` and `d_var` are local variables since they appear on the left of an equals sign in the function in the statements `b_var = 100 + a_var` and `d_var = 2 * a_var` .

Inside of the function `a_var` has no value assigned to it. When the function is called with `c_var = a_func(b_var)`, 15 is assigned to `a_var` since at that point in time `b_var` is 15, making the call to the function `a_func(15)`. This ends up setting `a_var` to 15 when it is inside of `a_func`.

As you can see, once the function finishes running, the local variables `a_var` and `b_var` that had hidden the global variables of the same name are gone. Then the statement `print("a_var = ", a_var)` prints the value `10` rather than the value `15` since the local variable that hid the global variable is gone.

Another thing to notice is the `NameError` that happens at the end. This appears since the variable `d_var` no longer exists since `a_func` finished. All the local variables are deleted when the function exits. If you want to get something from a function, then you will have to use `return something`.

One last thing to notice is that the value of `e_var` remains unchanged inside `a_func` since it is not a parameter and it never appears on the left of an equals sign inside of the function `a_func`. When a global variable is accessed inside a function it is the global variable from the outside.

Functions allow local variables that exist only inside the function and can hide other variables that are outside the function.

Examples

temperature2.py

```
#! /usr/bin/python
#-*-coding: utf-8 -*-
# converts temperature to Fahrenheit or Celsius

def print_options():
    print("Options:")
    print(" 'p' print options")
    print(" 'c' convert from Celsius")
    print(" 'f' convert from Fahrenheit")
    print(" 'q' quit the program")

def celsius_to_fahrenheit(c_temp):
    return 9.0 / 5.0 * c_temp + 32

def fahrenheit_to_celsius(f_temp):
    return (f_temp - 32.0) * 5.0 / 9.0

choice = "p"
while choice != "q":
    if choice == "c":
        c_temp = float(input("Celsius temperature: "))
        print("Fahrenheit:", celsius_to_fahrenheit(c_temp))
        choice = input("option: ")
    elif choice == "f":
        f_temp = float(input("Fahrenheit temperature: "))
        print("Celsius:", fahrenheit_to_celsius(f_temp))
        choice = input("option: ")
    elif choice == "p": #Alternatively choice != "q": so that print
                        #when anything unexpected inputed
        print_options()
        choice = input("option: ")
```

Sample Run:

```
Options:
 'p' print options
 'c' convert from celsius
 'f' convert from fahrenheit
 'q' quit the program
option: c
Celsius temperature: 30
Fahrenheit: 86.0
option: f
Fahrenheit temperature: 60
Celsius: 15.5555555556
option: q
```

area2.py

```
#! /usr/bin/python
#-*-coding: utf-8 -*-
# calculates a given rectangle area

def hello():
    print('Hello!')

def area(width, height):
    return width * height

def print_welcome(name):
    print('Welcome,', name)

def positive_input(prompt):
    number = float(input(prompt))
    while number <= 0:
        print('Must be a positive number')
        number = float(input(prompt))
    return number

name = input('Your Name: ')
hello()
print_welcome(name)
print()
print('To find the area of a rectangle,')
print('enter the width and height below.')
print()
w = positive_input('Width: ')
h = positive_input('Height: ')

print('Width =', w, ' Height =', h, ' so Area =', area(w, h))
```

Sample Run:

```
Your Name: Josh
Hello!
Welcome, Josh

To find the area of a rectangle,
enter the width and height below.

Width: -4
Must be a positive number
Width: 4
Height: 3
Width = 4  Height = 3  so Area = 12
```

Exercises

Rewrite the area2.py program from the Examples above to have a separate function for the area of a square, the area of a rectangle, and the area of a circle (`3.14 * radius**2`). This program should include a menu interface.

Solution

```
def square(L):
    return L * L

def rectangle(width , height):
    return width * height

def circle(radius):
    return 3.14159 * radius ** 2

def options():
    print()
    print("Options:")
    print("s = calculate the area of a square.")
    print("c = calculate the area of a circle.")
    print("r = calculate the area of a rectangle.")
    print("q = quit")
    print()

print("This program will calculate the area of a square, circle or rectangle.")
choice = "x"
options()
while choice != "q":
    choice = input("Please enter your choice: ")
    if choice == "s":
        L = float(input("Length of square side: "))
        print("The area of this square is", square(L))
        options()
    elif choice == "c":
        radius = float(input("Radius of the circle: "))
        print("The area of the circle is", circle(radius))
        options()
    elif choice == "r":
        width = float(input("Width of the rectangle: "))
        height = float(input("Height of the rectangle: "))
        print("The area of the rectangle is", rectangle(width, height))
        options()
    elif choice == "q":
        print(" ",end="")
    else:
        print("Unrecognized option.")
        options()
```

---

9. Advanced Functions Example

Some people find this section useful, and some find it confusing. If you find it confusing you can skip it. Now we will do a walk through for the following program:

```
def mult(a, b):
    if b == 0:
        return 0
    rest = mult(a, b - 1)
    value = a + rest
    return value
result = mult(3, 2)
print("3 * 2 = ", result)
```

Basically this program creates a positive integer multiplication function (that is far slower than the built in multiplication function) and then demonstrates this function with a use of the function. This program demonstrates the use of recursion, that is a form of iteration (repetition) in which there is a function that repeatedly calls itself until an exit condition is satisfied. It uses repeated additions to give the same result as mutiplication: e.g. 3 + 3 (addition) gives the same result as 3 * 2 (multiplication).

Question: **What is the first thing the program does?**

Answer: The first thing done is the function mult is defined with the lines:

```
def mult(a, b):
    if b == 0:
        return 0
    rest = mult(a, b - 1)
    value = a + rest
    return value
```

This creates a function that takes two parameters and returns a value when it is done. Later this function can be run.

What happens next?

The next line after the function, `result = mult(3, 2)` is run.

What does this line do?

This line will assign the return value of `mult(3, 2)` to the variable `result`.

And what does `mult(3, 2)` return?

We need to do a walkthrough of the `mult` function to find out.

What happens next?

The variable `a` gets the value 3 assigned to it and the variable `b` gets the value 2 assigned to it.

And then?

The line `if b == 0:` is run. Since `b` has the value 2 this is false so the line `return 0` is skipped.

And what then?

The line `rest = mult(a, b - 1)` is run. This line sets the local variable `rest` to the value of `mult(a, b - 1)`. The value of `a` is 3 and the value of `b` is 2 so the function call is `mult(3,1)`

So what is the value of `mult(3, 1)` ?

We will need to run the function `mult` with the parameters 3 and 1.

So what happens next?

The local variables in the *new* run of the function are set so that `a` has the value 3 and `b` has the value 1. Since these are local values these do not affect the previous values of `a` and `b`.

And then?

Since `b` has the value 1 the if statement is false, so the next line becomes `rest = mult(a, b - 1)`.

What does this line do?

This line will assign the value of `mult(3, 0)` to rest.

So what is that value?

We will have to run the function one more time to find that out. This time `a` has the value 3 and `b` has the value 0.

So what happens next?

The first line in the function to run is `if b == 0:`. `b` has the value 0 so the next line to run is `return 0`

And what does the line `return 0` do?

This line returns the value 0 out of the function.

So?

So now we know that `mult(3, 0)` has the value 0. Now we know what the line `rest = mult(a, b - 1)` did since we have run the function `mult` with the parameters 3 and 0. We have finished running `mult(3, 0)` and are now back to running `mult(3, 1)`. The variable `rest` gets assigned the value 0.

What line is run next?

The line `value = a + rest` is run next. In this run of the function, `a = 3` and `rest = 0` so now `value = 3`.

What happens next?

The line `return value` is run. This returns 3 from the function. This also exits from the run of the function `mult(3, 1)`. After `return` is called, we go back to running `mult(3, 2)`.

Where were we in `mult(3, 2)`?

We had the variables `a = 3` and `b = 2` and were examining the line `rest = mult(a, b - 1)`.

So what happens now?

The variable `rest` get 3 assigned to it. The next line `value = a + rest` sets `value` to `3 + 3` or 6.

So now what happens?

The next line runs, this returns 6 from the function. We are now back to running the line `result = mult(3, 2)` which can now assign the value 6 to the variable `result`.

What happens next?

The next line after the function, `print("3 * 2 = ", result)` is run.

And what does this do?

It prints `3 * 2 =` and the value of `result` which is 6. The complete line printed is `3 * 2 = 6`.

What is happening overall?

Basically we used two facts to calculate the multiple of the two numbers. The first is that any number times 0 is 0 (`x * 0 = 0`). The second is that a number times another number is equal to the first number plus the first number times one less than the second number (`x * y = x + x * (y - 1)`). So what happens is `3 * 2` is first converted into `3 + 3 * 1`. Then `3 * 1` is converted into `3 + 3 * 0`. Then we know that any number times 0 is 0 so `3 * 0` is 0. Then we can calculate that `3 + 3 * 0` is `3 + 0` which is `3`. Now we know what `3 * 1` is so we can calculate that `3 + 3 * 1` is `3 + 3` which is `6`.

This is how the whole thing works:

```
mult(3, 2)
3 + mult(3, 1)
3 + 3 + mult(3, 0)
3 + 3 + 0
3 + 3
6
```

Recursion

Programming constructs solving a problem by solving a smaller version of the same problem are called *recursive*. In the examples in this chapter, recursion is realized by defining a function calling itself. This facilitates implementing solutions to programming tasks as it may be sufficient to consider the next step of a problem instead of the whole problem at once. It is also useful as it allows to express some mathematical concepts with straightforward, easy to read code.

Any problem that can be solved with recursion could be re-implemented with loops. Using the latter usually results in better performance. However equivalent implementations using loops are usually harder to get done correctly.

Probably the most intuitive definition of *recursion* is:

Recursion

If you still don't get it, see *recursion*.

Try walking through the factorial example if the multiplication example did not make sense.

Examples

factorial.py

```
#defines a function that calculates the factorial
```

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print("2! =", factorial(2))
print("3! =", factorial(3))
print("4! =", factorial(4))
print("5! =", factorial(5))
```

Output:

```
2! = 2
3! = 6
4! = 24
5! = 120
```

countdown.py

```
def count_down(n):
    print(n)
    if n > 0:
        return count_down(n-1)

count_down(5)
```

Output:

```
5
4
3
2
1
0
```

---

10. Lists

Variables with more than one value

You have already seen ordinary variables that store a single value. However other variable types can hold more than one value. These are called containers because they can contain more than one object. The simplest type is called a list. Here is an example of a list being used:

```
which_one = int(input("What month (1-12)? "))
months = ['January', 'February', 'March', 'April', 'May', 'June', 'July',
          'August', 'September', 'October', 'November', 'December']

if 1 <= which_one <= 12:
    print("The month is", months[which_one - 1])
```

and an output example:

```
What month (1-12)? 3
The month is March
```

In this example the `months` is a list. `months` is defined with the lines `months = ['January', 'February', 'March', 'April', 'May', 'June', 'July',` and `'August', 'September', 'October', 'November', 'December']` (note that a `\` could also be used to split a long line, but that is not necessary in this case because Python is intelligent enough to recognize that everything within brackets belongs together). The `[` and `]` start and end the list with commas (`,`) separating the list items. The list is used in `months[which_one - 1]`. A list consists of items that are numbered starting at 0. In other words if you wanted January you would use `months[0]`. Give a list a number and it will return the value that is stored at that location.

The statement `if 1 <= which_one <= 12:` will only be true if `which_one` is between one and twelve inclusive (in other words it is what you would expect if you have seen that in algebra).

Lists can be thought of as a series of boxes. Each box has a different value. For example, the boxes created by `demolist = ['life', 42, 'the universe', 6, 'and', 9]` would look like this:

box number	**0**	**1**	**2**	**3**	**4**	**5**
demolist	"life"	42	"the universe"	6	"and"	9

Each box is referenced by its number so the statement `demolist[0]` would get `'life'`, `demolist[1]` would get `42` and so on up to `demolist[5]` getting `9`.

More features of lists

The next example is just to show a lot of other stuff lists can do (for once I don't expect you to type it in, but you should probably play around with lists in interactive mode until you are comfortable with them.). Here goes:

```
demolist = ["life", 42, "the universe", 6, "and", 9]
print("demolist = ",demolist)
demolist.append("everything")
print("after 'everything' was appended demolist is now:")
print(demolist)
print("len(demolist) =", len(demolist))
print("demolist.index(42) =", demolist.index(42))
print("demolist[1] =", demolist[1])

# Next we will loop through the list
for c in range(len(demolist)):
    print("demolist[", c, "] =", demolist[c])

del demolist[2]
print("After 'the universe' was removed demolist is now:")
print(demolist)
if "life" in demolist:
    print("'life' was found in demolist")
else:
    print("'life' was not found in demolist")

if "amoeba" in demolist:
    print("'amoeba' was found in demolist")

if "amoeba" not in demolist:
    print("'amoeba' was not found in demolist")

another_list = [42,7,0,123]
another_list.sort()
print("The sorted another_list is", another_list)
```

The output is:

```
demolist =  ['life', 42, 'the universe', 6, 'and', 9]
```

```
after 'everything' was appended demolist is now:
['life', 42, 'the universe', 6, 'and', 9, 'everything']
len(demolist) = 7
demolist.index(42) = 1
demolist[1] = 42
demolist[ 0 ] = life
demolist[ 1 ] = 42
demolist[ 2 ] = the universe
demolist[ 3 ] = 6
demolist[ 4 ] = and
demolist[ 5 ] = 9
demolist[ 6 ] = everything
After 'the universe' was removed demolist is now:
['life', 42, 6, 'and', 9, 'everything']
'life' was found in demolist
'amoeba' was not found in demolist
The sorted another_list is [0, 7, 42, 123]
```

This example uses a whole bunch of new functions. Notice that you can just `print` a whole list. Next the `append` function is used to add a new item to the end of the list. `len` returns how many items are in a list. The valid indexes (as in numbers that can be used inside of the `[]`) of a list range from 0 to `len - 1`. The `index` function tells where the first location of an item is located in a list. Notice how `demolist.index(42)` returns 1, and when `demolist[1]` is run it returns 42. To get help on all the functions a list provides for you, type `help(list)` in the interactive Python interpreter.

The line `# Next we will loop through the list` is a just a reminder to the programmer (also called a *comment*). Python ignores everything that is written after a `#` on the current line. Next the lines:

```
for c in range(len(demolist)):
    print('demolist[', c, '] =', demolist[c])
```

create a variable `c`, which starts at 0 and is incremented until it reaches the last index of the list. Meanwhile the `print` statement prints out each element of the list.

A much better way to do the above is:

```
for c, x in enumerate(demolist):
    print("demolist[", c, "] =", x)
```

The `del` command can be used to remove a given element in a list. The next few lines use the `in` operator to test if an element is in or is not in a list. The `sort` function sorts the list. This is useful if you need a list in order from smallest number to largest or alphabetical. Note that this rearranges the list. In summary, for a list, the following operations occur:

example	explanation
`demolist[2]`	accesses the element at index 2
`demolist[2] = 3`	sets the element at index 2 to be 3
`del demolist[2]`	removes the element at index 2
`len(demolist)`	returns the length of `demolist`
`"value" in demolist`	is *True* if `"value"` is an element in `demolist`
`"value" not in demolist`	is *True* if `"value"` is not an element in `demolist`
`another_list.sort()`	sorts `another_list`. Note that the list must be all numbers or all strings to be sorted.
`demolist.index("value")`	returns the index of the first place that `"value"` occurs
`demolist.append("value")`	adds an element `"value"` at the end of the list
`demolist.remove("value")`	removes the first occurrence of value from `demolist` (same as `del demolist[demolist.index("value")]`)

This next example uses these features in a more useful way:

```
menu_item = 0
namelist = []
while menu_item != 9:
    print("--------------------")
    print("1. Print the list")
    print("2. Add a name to the list")
    print("3. Remove a name from the list")
    print("4. Change an item in the list")
    print("9. Quit")
    menu_item = int(input("Pick an item from the menu: "))
    if menu_item == 1:
        current = 0
        if len(namelist) > 0:
            while current < len(namelist):
                print(current, ".", namelist[current])
                current = current + 1
        else:
            print("List is empty")
    elif menu_item == 2:
        name = input("Type in a name to add: ")
        namelist.append(name)
    elif menu_item == 3:
        del_name = input("What name would you like to remove: ")
        if del_name in namelist:
            # namelist.remove(del_name) would work just as fine
            item_number = namelist.index(del_name)
            del namelist[item_number]
            # The code above only removes the first occurrence of
            # the name.  The code below from Gerald removes all.
            # while del_name in namelist:
            #       item_number = namelist.index(del_name)
            #       del namelist[item_number]
        else:
            print(del_name, "was not found")
    elif menu_item == 4:
        old_name = input("What name would you like to change: ")
        if old_name in namelist:
            item_number = namelist.index(old_name)
            new_name = input("What is the new name: ")
            namelist[item_number] = new_name
        else:
            print(old_name, "was not found")

print("Goodbye")
```

And here is part of the output:

```
--------------------
1. Print the list
2. Add a name to the list
3. Remove a name from the list
4. Change an item in the list
9. Quit

Pick an item from the menu: 2
Type in a name to add: Jack

Pick an item from the menu: 2
Type in a name to add: Jill

Pick an item from the menu: 1
0 . Jack
1 . Jill

Pick an item from the menu: 3
What name would you like to remove: Jack

Pick an item from the menu: 4
What name would you like to change: Jill
What is the new name: Jill Peters

Pick an item from the menu: 1
0 . Jill Peters

Pick an item from the menu: 9
Goodbye
```

That was a long program. Let's take a look at the source code. The line `namelist = []` makes the variable `namelist` a list with no items (or elements). The next important line is `while menu_item != 9:`. This line starts a loop that allows the menu system for this program. The next few lines display a menu and decide which part of the program to run.

The section

```
current = 0
if len(namelist) > 0:
    while current < len(namelist):
        print(current, ".", namelist[current])
        current = current + 1
else:
    print("List is empty")
```

goes through the list and prints each name. `len(namelist)` tells how many items are in the list. If `len` returns `0`, then the list is empty.

Then, a few lines later, the statement `namelist.append(name)` appears. It uses the `append` function to add an item to the end of the list. Jump down another two lines, and notice this section of code:

```
item_number = namelist.index(del_name)
del namelist[item_number]
```

Here the `index` function is used to find the index value that will be used later to remove the item. `del namelist[item_number]` is used to remove a element of the list.

The next section

```
old_name = input("What name would you like to change: ")
if old_name in namelist:
    item_number = namelist.index(old_name)
    new_name = input("What is the new name: ")
```

```
    namelist[item_number] = new_name
else:
   print(old_name, "was not found")
```

uses `index` to find the `item_number` and then puts `new_name` where the `old_name` was.

Congratulations, with lists under your belt, you now know enough of the language that you could do any computations that a computer can do (this is technically known as Turing-Completeness). Of course, there are still many features that are used to make your life easier.

Examples

test.py

```
## This program runs a test of knowledge

# First get the test questions
# Later this will be modified to use file io.
def get_questions():
    # notice how the data is stored as a list of lists
    return [["What color is the daytime sky on a clear day? ", "blue"],
            ["What is the answer to life, the universe and everything? ", "42"],
            ["What is a three letter word for mouse trap? ", "cat"]]

# This will test a single question
# it takes a single question in
# it returns True if the user typed the correct answer, otherwise False

def check_question(question_and_answer):
    # extract the question and the answer from the list
    # This function takes a list with two elements, a question and an answer.
    question = question_and_answer[0]
    answer = question_and_answer[1]
    # give the question to the user
    given_answer = input(question)
    # compare the user's answer to the tester's answer
    if answer == given_answer:
        print("Correct")
        return True
    else:
        print("Incorrect, correct was:", answer)
        return False

# This will run through all the questions
def run_test(questions):
    if len(questions) == 0:
        print("No questions were given.")
        # the return exits the function
        return
    index = 0
    right = 0
    while index < len(questions):
        # Check the question
        #Note that this is extracting a question and answer list from the list of lists.
        if check_question(questions[index]):
            right = right + 1
        # go to the next question
        index = index + 1
    # notice the order of the computation, first multiply, then divide
    print("You got", right * 100 / len(questions),\
          "% right out of", len(questions))

# now let's get the questions from the get_questions function, and
# send the returned list of lists as an argument to the run_test function.

run_test(get_questions())
```

The values `True` and `False` point to 1 and 0, respectively. They are often used in sanity checks, loop conditions etc. You will learn more about this a little bit later (chapter Boolean Expressions). Please note that get_questions() is essentially a list because even though it's technically a function, returning a list of lists

is the only thing it does.

Sample Output:

```
What color is the daytime sky on a clear day? green
Incorrect, correct was: blue
What is the answer to life, the universe and everything? 42
Correct
What is a three letter word for mouse trap? cat
Correct
You got 66 % right out of 3
```

Exercises

Expand the test.py program so it has a menu giving the option of taking the test, viewing the list of questions and answers, and an option to quit. Also, add a new question to ask, "What noise does a truly advanced machine make?" with the answer of "ping".

Solution

Expand the test.py program so it has menu giving the option of taking the test, viewing the list of questions and answers, and an option to quit. Also, add a new question to ask, "What noise does a truly advanced machine make?" with the answer of "ping".

```
## This program runs a test of knowledge

questions = [["What color is the daytime sky on a clear day? ", "blue"],
             ["What is the answer to life, the universe and everything? ", "42"],
             ["What is a three letter word for mouse trap? ", "cat"],
             ["What noise does a truly advanced machine make?", "ping"]]

# This will test a single question
# it takes a single question in
# it returns True if the user typed the correct answer, otherwise False

def check_question(question_and_answer):
    # extract the question and the answer from the list
    question = question_and_answer[0]
    answer = question_and_answer[1]
    # give the question to the user
    given_answer = input(question)
    # compare the user's answer to the testers answer
    if answer == given_answer:
        print("Correct")
        return True
    else:
        print("Incorrect, correct was:", answer)
        return False

# This will run through all the questions

def run_test(questions):

    if len(questions) == 0:
        print("No questions were given.")
        # the return exits the function
        return
    index = 0
    right = 0
    while index < len(questions):
        # Check the question
        if check_question(questions[index]):
            right = right + 1
        # go to the next question
        index = index + 1
    # notice the order of the computation, first multiply, then divide
    print("You got", right * 100 / len(questions),
          "% right out of", len(questions))
```

```
#showing a list of questions and answers
def showquestions():
    q = 0
    while q < len(questions):
        a = 0
        print("Q:" , questions[q][a])
        a = 1
        print("A:" , questions[q][a])
        q = q + 1

# now let's define the menu function
def menu():
    print("-----------------")
    print("Menu:")
    print("1 - Take the test")
    print("2 - View a list of questions and answers")
    print("3 - View the menu")
    print("5 - Quit")
    print("-----------------")

choice = "3"
while choice != "5":
    if choice == "1":
        run_test(questions)
    elif choice == "2":
        showquestions()
    elif choice == "3":
        menu()
    print()
    choice = input("Choose your option from the menu above: ")
```

---

11. For Loops

And here is the new typing exercise for this chapter:

```
onetoten = range(1, 11)
for count in onetoten:
    print(count)
```

and the ever-present output:

```
1
2
3
4
5
6
7
8
9
10
```

The output looks awfully familiar but the program code looks different. The first line uses the `range` function. The `range` function uses two arguments like this `range(start, finish)`. `start` is the first number that is produced. `finish` is one larger than the last number. Note that this program could have been done in a shorter way:

```
for count in range(1, 11):
    print(count)
```

The range function returns an iterable. This can be converted into a list with the `list` function. which will

then be the dominant number. Here are some examples to show what happens with the `range` command:

```
>>> range(1, 10)
range(1, 10)
>>> list(range(1, 10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(-32, -20))
[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]
>>> list(range(5,21))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> list(range(21, 5))
[]
```

The next line `for count in onetoten:` uses the `for` control structure. A `for` control structure looks like `for variable in list:`. `list` is gone through starting with the first element of the list and going to the last. As `for` goes through each element in a list it puts each into `variable`. That allows `variable` to be used in each successive time the `for` loop is run through. Here is another example (you don't have to type this) to demonstrate:

```
demolist = ['life', 42, 'the universe', 6, 'and', 7, 'everything']
for item in demolist:
    print("The current item is:",item)
```

The output is:

```
The current item is: life
The current item is: 42
The current item is: the universe
The current item is: 6
The current item is: and
The current item is: 7
The current item is: everything
```

Notice how the `for` loop goes through and sets item to each element in the list. So, what is `for` good for? The first use is to go through all the elements of a list and do something with each of them. Here's a quick way to add up all the elements:

```
list = [2, 4, 6, 8]
sum = 0
for num in list:
    sum = sum + num

print("The sum is:", sum)
```

with the output simply being:

The sum is: 20

Or you could write a program to find out if there are any duplicates in a list like this program does:

```
list = [4, 5, 7, 8, 9, 1, 0, 7, 10]
list.sort()
prev = None
for item in list:
    if prev == item:
        print("Duplicate of", prev, "found")
    prev = item
```

and for good measure:

```
Duplicate of 7 found
```

Okay, so how does it work? Here is a special debugging version to help you understand (you don't need to type this in):

```
l = [4, 5, 7, 8, 9, 1, 0, 7, 10]
print("l = [4, 5, 7, 8, 9, 1, 0, 7, 10]", "\t\tl:", l)
l.sort()
print("l.sort()", "\t\tl:", l)
prev = l[0]
print("prev = l[0]", "\t\tprev:", prev)
del l[0]
print("del l[0]", "\t\tl:", l)
for item in l:
    if prev == item:
        print("Duplicate of", prev, "found")
    print("if prev == item:", "\t\tprev:", prev, "\titem:", item)
    prev = item
    print("prev = item", "\t\tprev:", prev, "\titem:", item)
```

with the output being:

```
l = [4, 5, 7, 8, 9, 1, 0, 7, 10]        l: [4, 5, 7, 8, 9, 1, 0, 7, 10]
l.sort()                l: [0, 1, 4, 5, 7, 7, 8, 9, 10]
prev = l[0]             prev: 0
del l[0]                l: [1, 4, 5, 7, 7, 8, 9, 10]
if prev == item:        prev: 0         item: 1
prev = item             prev: 1         item: 1
if prev == item:        prev: 1         item: 4
prev = item             prev: 4         item: 4
if prev == item:        prev: 4         item: 5
prev = item             prev: 5         item: 5
if prev == item:        prev: 5         item: 7
prev = item             prev: 7         item: 7
Duplicate of 7 found
if prev == item:        prev: 7         item: 7
prev = item             prev: 7         item: 7
if prev == item:        prev: 7         item: 8
prev = item             prev: 8         item: 8
if prev == item:        prev: 8         item: 9
prev = item             prev: 9         item: 9
if prev == item:        prev: 9         item: 10
prev = item             prev: 10        item: 10
```

The reason I put so many `print` statements in the code was so that you can see what is happening in each line. (By the way, if you can't figure out why a program is not working, try putting in lots of print statements in places where you want to know what is happening.) First the program starts with a boring old list. Next the program sorts the list. This is so that any duplicates get put next to each other. The program then initializes a `prev`(ious) variable. Next the first element of the list is deleted so that the first item is not incorrectly thought to be a duplicate. Next a `for` loop is gone into. Each item of the list is checked to see if it is the same as the previous. If it is a duplicate was found. The value of `prev` is then changed so that the next time the `for` loop is run through `prev` is the previous item to the current. Sure enough, the 7 is found to be a duplicate. (Notice how `\t` is used to print a tab.)

The other way to use `for` loops is to do something a certain number of times. Here is some code to print out the first 9 numbers of the Fibonacci series:

```
a = 1
b = 1
for c in range(1, 10):
```

```
    print(a, end=" ")
    n = a + b
    a = b
    b = n
```

with the surprising output:

```
1 1 2 3 5 8 13 21 34
```

Everything that can be done with `for` loops can also be done with `while` loops but `for` loops give an easy way to go through all the elements in a list or to do something a certain number of times.

---

12. Boolean Expressions

Here is a little example of boolean expressions (you don't have to type it in):

```
a = 6
b = 7
c = 42
print(1, a == 6)
print(2, a == 7)
print(3, a == 6 and b == 7)
print(4, a == 7 and b == 7)
print(5, not a == 7 and b == 7)
print(6, a == 7 or b == 7)
print(7, a == 7 or b == 6)
print(8, not (a == 7 and b == 6))
print(9, not a == 7 and b == 6)
```

With the output being:

```
1 True
2 False
3 True
4 False
5 True
6 True
7 False
8 True
9 False
```

What is going on? The program consists of a bunch of funny looking `print` statements. Each `print` statement prints a number and an expression. The number is to help keep track of which statement I am dealing with. Notice how each expression ends up being either `False` or `True`. In Python false can also be written as 0 and true as 1.

The lines:

```
print(1, a == 6)
print(2, a == 7)
```

print out a `True` and a `False` respectively just as expected since the first is true and the second is false. The third print, `print(3, a == 6 and b == 7)`, is a little different. The operator `and` means if both the statement before and the statement after are true then the whole expression is true otherwise the whole expression is false. The next line, `print(4, a == 7 and b == 7)`, shows how if part of an `and` expression is false, the whole thing is false. The behavior of `and` can be summarized as follows:

expression	result
true `and` true	true
true `and` false	false
false `and` true	false
false `and` false	false

Notice that if the first expression is false Python does not check the second expression since it knows the whole expression is false. Try running `False and print("Hi")` and compare this to running `True and print("Hi")` The technical term for this is short-circuit evaluation

The next line, `print(5, not a == 7 and b == 7)`, uses the `not` operator. `not` just gives the opposite of the expression. (The expression could be rewritten as `print(5, a != 7 and b == 7)`). Here is the table:

expression	result
`not` true	false
`not` false	true

The two following lines, `print(6, a == 7 or b == 7)` and `print(7, a == 7 or b == 6)`, use the `or` operator. The `or` operator returns true if the first expression is true, or if the second expression is true or both are true. If neither are true it returns false. Here's the table:

expression	result
true `or` true	true
true `or` false	true
false `or` true	true
false `or` false	false

Notice that if the first expression is true Python doesn't check the second expression since it knows the whole expression is true. This works since `or` is true if at least one half of the expression is true. The first part is true so the second part could be either false or true, but the whole expression is still true.

The next two lines, `print(8, not (a == 7 and b == 6))` and `print(9, not a == 7 and b == 6)`, show that parentheses can be used to group expressions and force one part to be evaluated first. Notice that the parentheses changed the expression from false to true. This occurred since the parentheses forced the `not` to apply to the whole expression instead of just the `a == 7` portion.

Here is an example of using a boolean expression:

```
list = ["Life", "The Universe", "Everything", "Jack", "Jill", "Life", "Jill"]

# make a copy of the list. See the More on Lists chapter to explain what [:] means.
copy = list[:]
# sort the copy
copy.sort()
prev = copy[0]
del copy[0]

count = 0

# go through the list searching for a match
while count < len(copy) and copy[count] != prev:
```

```
    prev = copy[count]
    count = count + 1

# If a match was not found then count can't be < len
# since the while loop continues while count is < len
# and no match is found

if count < len(copy):
    print("First Match:", prev)
```

And here is the output:

```
First Match: Jill
```

This program works by continuing to check for match `while count < len(copy) and copy[count] is not equal to prev`. When either `count` is greater than the last index of `copy` or a match has been found the `and` is no longer true so the loop exits. The `if` simply checks to make sure that the `while` exited because a match was found.

The other "trick" of `and` is used in this example. If you look at the table for `and` notice that the third entry is "false and false". If `count >= len(copy)` (in other words `count < len(copy)` is false) then `copy[count]` is never looked at. This is because Python knows that if the first is false then they can't both be true. This is known as a short circuit and is useful if the second half of the `and` will cause an error if something is wrong. I used the first expression (`count < len(copy)`) to check and see if `count` was a valid index for `copy`. (If you don't believe me remove the matches "Jill" and "Life", check that it still works and then reverse the order of `count < len(copy) and copy[count] != prev` to `copy[count] != prev and count < len(copy)`.)

Boolean expressions can be used when you need to check two or more different things at once.

A note on Boolean Operators

A common mistake for people new to programming is a misunderstanding of the way that boolean operators works, which stems from the way the python interpreter reads these expressions. For example, after initially learning about "and " and "or" statements, one might assume that the expression `x == ('a' or 'b')` would check to see if the variable `x` was equivalent to one of the strings `'a'` or `'b'`. This is not so. To see what I'm talking about, start an interactive session with the interpreter and enter the following expressions:

```
>>> 'a' == ('a' or 'b')
>>> 'b' == ('a' or 'b')
>>> 'a' == ('a' and 'b')
>>> 'b' == ('a' and 'b')
```

And this will be the unintuitive result:

```
>>> 'a' == ('a' or 'b')
True
>>> 'b' == ('a' or 'b')
False
>>> 'a' == ('a' and 'b')
False
>>> 'b' == ('a' and 'b')
True
```

At this point, the `and` and `or` operators seem to be broken. It doesn't make sense that, for the first two expressions, `'a'` is equivalent to `'a'` or `'b'` while `'b'` is not. Furthermore, it doesn't make any sense that 'b'

is equivalent to `'a'` and `'b'`. After examining what the interpreter does with boolean operators, these results do in fact exactly what you are asking of them, it's just not the same as what you think you are asking.

When the Python interpreter looks at an `or` expression, it takes the first statement and checks to see if it is true. If the first statement is true, then Python returns that object's value without checking the second statement. This is because for an `or` expression, the whole thing is true if one of the values is true; the program does not need to bother with the second statement. On the other hand, if the first value is evaluated as false Python checks the second half and returns that value. That second half determines the truth value of the whole expression since the first half was false. This "laziness" on the part of the interpreter is called "short circuiting" and is a common way of evaluating boolean expressions in many programming languages.

Similarly, for an `and` expression, Python uses a short circuit technique to speed truth value evaluation. If the first statement is false then the whole thing must be false, so it returns that value. Otherwise if the first value is true it checks the second and returns that value.

One thing to note at this point is that the boolean expression returns a value indicating `True` or `False`, but that Python considers a number of different things to have a truth value assigned to them. To check the truth value of any given object `x`, you can use the fuction `bool(x)` to see its truth value. Below is a table with examples of the truth values of various objects:

True	False
True	False
1	0
Numbers other than zero	The string 'None'
Nonempty strings	Empty strings
Nonempty lists	Empty lists
Nonempty dictionaries	Empty dictionaries

Now it is possible to understand the perplexing results we were getting when we tested those boolean expressions before. Let's take a look at what the interpreter "sees" as it goes through that code:

First case:

```
>>> 'a' == ('a' or 'b')  # Look at parentheses first, so evaluate expression "('a' or 'b')"
                         # 'a' is a nonempty string, so the first value is True
                         # Return that first value: 'a'
>>> 'a' == 'a'          # the string 'a' is equivalent to the string 'a', so expression is True
True
```

Second case:

```
>>> 'b' == ('a' or 'b')  # Look at parentheses first, so evaluate expression "('a' or 'b')"
                         # 'a' is a nonempty string, so the first value is True
                         # Return that first value: 'a'
>>> 'b' == 'a'          # the string 'b' is not equivalent to the string 'a', so expression is False
False
```

Third case:

```
>>> 'a' == ('a' and 'b') # Look at parentheses first, so evaluate expression "('a' and 'b')"
                         # 'a' is a nonempty string, so the first value is True, examine second value
                         # 'b' is a nonempty string, so second value is True
```

```
                          # Return that second value as result of whole expression: 'b'
>>> 'a' == 'b'           # the string 'a' is not equivalent to the string 'b', so expression is False
False
```

Fourth case:

```
>>> 'b' == ('a' and 'b') # Look at parentheses first, so evaluate expression "('a' and 'b')"
                          # 'a' is a nonempty string, so the first value is True, examine second value
                          # 'b' is a nonempty string, so second value is True
                          # Return that second value as result of whole expression: 'b'
>>> 'b' == 'b'           # the string 'b' is equivalent to the string 'b', so expression is True
True
```

So Python was really doing its job when it gave those apparently bogus results. As mentioned previously, the important thing is to recognize what value your boolean expression will return when it is evaluated, because it isn't always obvious.

Going back to those initial expressions, this is how you would write them out so they behaved in a way that you want:

```
>>> 'a' == 'a' or 'a' == 'b'
True
>>> 'b' == 'a' or 'b' == 'b'
True
>>> 'a' == 'a' and 'a' == 'b'
False
>>> 'b' == 'a' and 'b' == 'b'
False
```

When these comparisons are evaluated they return truth values in terms of True or False, not strings, so we get the proper results.

Examples

password1.py

```
## This program asks a user for a name and a password.
# It then checks them to make sure that the user is allowed in.

name = input("What is your name? ")
password = input("What is the password? ")
if name == "Josh" and password == "Friday":
    print("Welcome Josh")
elif name == "Fred" and password == "Rock":
    print("Welcome Fred")
else:
    print("I don't know you.")
```

Sample runs

```
What is your name? Josh
What is the password? Friday
Welcome Josh
```

```
What is your name? Bill
What is the password? Money
I don't know you.
```

Exercises

Write a program that has a user guess your name, but they only get 3 chances to do so until the program quits.

Solution

```
print("Try to guess my name!")
count = 1
name = "guilherme"
guess = input("What is my name? ")
while count < 3 and guess.lower() != name:    # .lower allows things like Guilherme to still match
    print("You are wrong!")
    guess = input("What is my name? ")
    count = count + 1

if guess.lower() != name:
    print("You are wrong!") # this message isn't printed in the third chance, so we print it now
    print("You ran out of chances.")
else:
    print("Yes! My name is", name + "!")
```

---

13. Dictionaries

This chapter is about dictionaries. Dictionaries have keys and values. The keys are used to find the values. Here is an example of a dictionary in use:

```
def print_menu():
    print('1. Print Phone Numbers')
    print('2. Add a Phone Number')
    print('3. Remove a Phone Number')
    print('4. Lookup a Phone Number')
    print('5. Quit')
    print()

numbers = {}
menu_choice = 0
print_menu()
while menu_choice != 5:
    menu_choice = int(input("Type in a number (1-5): "))
    if menu_choice == 1:
        print("Telephone Numbers:")
        for x in numbers.keys():
            print("Name: ", x, "\tNumber:", numbers[x])
        print()
    elif menu_choice == 2:
        print("Add Name and Number")
        name = input("Name: ")
        phone = input("Number: ")
        numbers[name] = phone
    elif menu_choice == 3:
        print("Remove Name and Number")
        name = input("Name: ")
        if name in numbers:
            del numbers[name]
        else:
            print(name, "was not found")
    elif menu_choice == 4:
        print("Lookup Number")
        name = input("Name: ")
        if name in numbers:
            print("The number is", numbers[name])
        else:
            print(name, "was not found")
    elif menu_choice != 5:
        print_menu()
```

And here is my output:

```
1. Print Phone Numbers
2. Add a Phone Number
3. Remove a Phone Number
4. Lookup a Phone Number
5. Quit

Type in a number (1-5): 2
Add Name and Number
Name: Joe
Number: 545-4464
Type in a number (1-5): 2
Add Name and Number
Name: Jill
Number: 979-4654
Type in a number (1-5): 2
Add Name and Number
Name: Fred
Number: 132-9874
Type in a number (1-5): 1
Telephone Numbers:
Name: Jill      Number: 979-4654
Name: Joe       Number: 545-4464
Name: Fred      Number: 132-9874

Type in a number (1-5): 4
Lookup Number
Name: Joe
The number is 545-4464
Type in a number (1-5): 3
Remove Name and Number
Name: Fred
Type in a number (1-5): 1
Telephone Numbers:
Name: Jill      Number: 979-4654
Name: Joe       Number: 545-4464

Type in a number (1-5): 5
```

This program is similar to the name list earlier in the chapter on lists. Here's how the program works. First the function `print_menu` is defined. `print_menu` just prints a menu that is later used twice in the program. Next comes the funny looking line `numbers = {}`. All that this line does is to tell Python that `numbers` is a dictionary. The next few lines just make the menu work. The lines

```
for x in numbers.keys():
    print("Name:", x, "\tNumber:", numbers[x])
```

go through the dictionary and print all the information. The function `numbers.keys()` returns a list that is then used by the `for` loop. The list returned by `keys()` is not in any particular order so if you want it in alphabetic order it must be sorted. Similar to lists the statement `numbers[x]` is used to access a specific member of the dictionary. Of course in this case `x` is a string. Next the line `numbers[name] = phone` adds a name and phone number to the dictionary. If `name` had already been in the dictionary `phone` would replace whatever was there before. Next the lines

```
if name in numbers:
    del numbers[name]
```

see if a name is in the dictionary and remove it if it is. The operator `name in numbers` returns true if `name` is in `numbers` but otherwise returns false. The line `del numbers[name]` removes the key `name` and the value associated with that key. The lines

```
if name in numbers:
```

```
    print("The number is", numbers[name])
```

check to see if the dictionary has a certain key and if it does prints out the number associated with it. Lastly if the menu choice is invalid it reprints the menu for your viewing pleasure.

A recap: Dictionaries have keys and values. Keys can be strings or numbers. Keys point to values. Values can be any type of variable (including lists or even dictionaries (those dictionaries or lists of course can contain dictionaries or lists themselves (scary right? :-)))). Here is an example of using a list in a dictionary:

```
max_points = [25, 25, 50, 25, 100]
assignments = ['hw ch 1', 'hw ch 2', 'quiz   ', 'hw ch 3', 'test']
students = {'#Max': max_points}

def print_menu():
    print("1. Add student")
    print("2. Remove student")
    print("3. Print grades")
    print("4. Record grade")
    print("5. Print Menu")
    print("6. Exit")

def print_all_grades():
    print('\t', end=' ')
    for i in range(len(assignments)):
        print(assignments[i], '\t', end=' ')
    print()
    keys = list(students.keys())
    keys.sort()
    for x in keys:
        print(x, '\t', end=' ')
        grades = students[x]
        print_grades(grades)

def print_grades(grades):
    for i in range(len(grades)):
        print(grades[i], '\t', end=' ')
    print()

print_menu()
menu_choice = 0
while menu_choice != 6:
    print()
    menu_choice = int(input("Menu Choice (1-6): "))
    if menu_choice == 1:
        name = input("Student to add: ")
        students[name] = [0] * len(max_points)
    elif menu_choice == 2:
        name = input("Student to remove: ")
        if name in students:
            del students[name]
        else:
            print("Student:", name, "not found")
    elif menu_choice == 3:
        print_all_grades()
    elif menu_choice == 4:
        print("Record Grade")
        name = input("Student: ")
        if name in students:
            grades = students[name]
            print("Type in the number of the grade to record")
            print("Type a 0 (zero) to exit")
            for i in range(len(assignments)):
                print(i + 1, assignments[i], '\t', end=' ')
            print()
            print_grades(grades)
            which = 1234
            while which != -1:
                which = int(input("Change which Grade: "))
                which -= 1    #same as which = which - 1
                if 0 <= which < len(grades):
                    grade = int(input("Grade: "))
                    grades[which] = grade
                elif which != -1:
                    print("Invalid Grade Number")
```

```
        else:
            print("Student not found")
    elif menu_choice != 6:
        print_menu()
```

and here is a sample output:

```
1. Add student
2. Remove student
3. Print grades
4. Record grade
5. Print Menu
6. Exit

Menu Choice (1-6): 3
       hw ch 1          hw ch 2          quiz             hw ch 3          test
#Max    25               25               50               25               100

Menu Choice (1-6): 5
1. Add student
2. Remove student
3. Print grades
4. Record grade
5. Print Menu
6. Exit

Menu Choice (1-6): 1
Student to add: Bill

Menu Choice (1-6): 4
Record Grade
Student: Bill
Type in the number of the grade to record
Type a 0 (zero) to exit
1   hw ch 1     2   hw ch 2     3   quiz        4   hw ch 3     5   test
0               0               0               0               0
Change which Grade: 1
Grade: 25
Change which Grade: 2
Grade: 24
Change which Grade: 3
Grade: 45
Change which Grade: 4
Grade: 23
Change which Grade: 5
Grade: 95
Change which Grade: 0

Menu Choice (1-6): 3
       hw ch 1          hw ch 2          quiz             hw ch 3          test
#Max    25               25               50               25               100
Bill    25               24               45               23               95

Menu Choice (1-6): 6
```

Heres how the program works. Basically the variable `students` is a dictionary with the keys being the name of the students and the values being their grades. The first two lines just create two lists. The next line `students = {'#Max': max_points}` creates a new dictionary with the key `{#Max}` and the value is set to be `[25, 25, 50, 25, 100]` (since thats what `max_points` was when the assignment is made) (I use the key `#Max` since `#` is sorted ahead of any alphabetic characters). Next `print_menu` is defined. Next the `print_all_grades` function is defined in the lines:

```
def print_all_grades():
    print('\t',end=" ")
    for i in range(len(assignments)):
        print(assignments[i], '\t',end=" ")
    print()
    keys = list(students.keys())
    keys.sort()
    for x in keys:
```

```
        print(x, '\t',end=' ')
        grades = students[x]
        print_grades(grades)
```

Notice how first the keys are gotten out of the `students` dictionary with the `keys` function in the line `keys = list(students.keys())`. `keys` is a iterable, and it is converted to list so all the functions for lists can be used on it. Next the keys are sorted in the line `keys.sort()`. `for` is used to go through all the keys. The grades are stored as a list inside the dictionary so the assignment `grades = students[x]` gives `grades` the list that is stored at the key `x`. The function `print_grades` just prints a list and is defined a few lines later.

The later lines of the program implement the various options of the menu. The line `students[name] = [0] * len(max_points)` adds a student to the key of their name. The notation `[0] * len(max_points)` just creates a list of 0's that is the same length as the `max_points` list.

The remove student entry just deletes a student similar to the telephone book example. The record grades choice is a little more complex. The grades are retrieved in the line `grades = students[name]` gets a reference to the grades of the student `name`. A grade is then recorded in the line `grades[which] = grade`. You may notice that `grades` is never put back into the students dictionary (as in no `students[name] = grades`). The reason for the missing statement is that `grades` is actually another name for `students[name]` and so changing `grades` changes `student[name]`.

Dictionaries provide a easy way to link keys to values. This can be used to easily keep track of data that is attached to various keys.

---

14. Using Modules

Here's this chapter's typing exercise (name it cal.py (`import` actually looks for a file named calendar.py and reads it in. If the file is named calendar.py and it sees a "import calendar" it tries to read in itself which works poorly at best.)):

```
import calendar
year = int(input("Type in the year number: "))
calendar.prcal(year)
```

And here is part of the output I got:

```
Type in the year number: 2001

                                  2001

      January                    February                   March

Mo Tu We Th Fr Sa Su       Mo Tu We Th Fr Sa Su       Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                1  2  3  4
 8  9 10 11 12 13 14        5  6  7  8  9 10 11        5  6  7  8  9 10 11
15 16 17 18 19 20 21       12 13 14 15 16 17 18       12 13 14 15 16 17 18
22 23 24 25 26 27 28       19 20 21 22 23 24 25       19 20 21 22 23 24 25
29 30 31                   26 27 28                   26 27 28 29 30 31
```

(I skipped some of the output, but I think you get the idea.) So what does the program do? The first line `import calendar` uses a new command `import`. The command `import` loads a module (in this case the `calendar` module). To see the commands available in the standard modules either look in the library reference for python (if you downloaded it) or go to http://docs.python.org/3/library/. If you look at the documentation for the calendar module, it lists a function called `prcal` that prints a calendar for a year. The

line `calendar.prcal(year)` uses this function. In summary to use a module `import` it and then use `module_name.function` for functions in the module. Another way to write the program is:

```
from calendar import prcal

year = int(input("Type in the year number: "))
prcal(year)
```

This version imports a specific function from a module. Here is another program that uses the Python Library (name it something like clock.py) (press Ctrl and the 'c' key at the same time to terminate the program):

```
from time import time, ctime

prev_time = ""
while True:
    the_time = ctime(time())
    if prev_time != the_time:
        print("The time is:", ctime(time()))
        prev_time = the_time
```

With some output being:

```
The time is: Sun Aug 20 13:40:04 2000
The time is: Sun Aug 20 13:40:05 2000
The time is: Sun Aug 20 13:40:06 2000
The time is: Sun Aug 20 13:40:07 2000

Traceback (innermost last):
 File "clock.py", line 5, in ?
    the_time = ctime(time())

KeyboardInterrupt
```

The output is infinite of course so I canceled it (or the output at least continues until Ctrl+C is pressed). The program just does a infinite loop (`True` is always true, so `while True:` goes forever) and each time checks to see if the time has changed and prints it if it has. Notice how multiple names after the import statement are used in the line `from time import time, ctime`.

The Python Library contains many useful functions. These functions give your programs more abilities and many of them can simplify programming in Python.

Exercises

Rewrite the `high_low.py` program from section Decisions to use an random integer between 0 and 99 instead of the hard-coded 78. Use the Python documentation to find an appropriate module and function to do this.

Solution

Rewrite the `high_low.py` program from section Decisions to use an random integer between 0 and 99 instead of the hard-coded 78. Use the Python documentation to find an appropriate module and function to do this.

```
from random import randint
number = randint(0, 99)
guess = -1
while guess != number:
```

```
    guess = int(input ("Guess a number: "))
    if guess > number:
        print("Too high")
    elif guess < number:
            print("Too low")
print("Just right")
```

---

15. More on Lists

We have already seen lists and how they can be used. Now that you have some more background I will go into more detail about lists. First we will look at more ways to get at the elements in a list and then we will talk about copying them.

Here are some examples of using indexing to access a single element of a list:

```
>>> some_numbers = ['zero', 'one', 'two', 'three', 'four', 'five']
>>> some_numbers[0]
'zero'
>>> some_numbers[4]
'four'
>>> some_numbers[5]
'five'
```

All those examples should look familiar to you. If you want the first item in the list just look at index 0. The second item is index 1 and so on through the list. However what if you want the last item in the list? One way could be to use the `len()` function like `some_numbers[len(some_numbers) - 1]`. This way works since the `len()` function always returns the last index plus one. The second from the last would then be `some_numbers[len(some_numbers) - 2]`. There is an easier way to do this. In Python the last item is always index -1. The second to the last is index -2 and so on. Here are some more examples:

```
>>> some_numbers[len(some_numbers) - 1]
'five'
>>> some_numbers[len(some_numbers) - 2]
'four'
>>> some_numbers[-1]
'five'
>>> some_numbers[-2]
'four'
>>> some_numbers[-6]
'zero'
```

Thus any item in the list can be indexed in two ways: from the front and from the back.

Another useful way to get into parts of lists is using slicing. Here is another example to give you an idea what they can be used for:

```
>>> things = [0, 'Fred', 2, 'S.P.A.M.', 'Stocking', 42, "Jack", "Jill"]
>>> things[0]
0
>>> things[7]
'Jill'
>>> things[0:8]
[0, 'Fred', 2, 'S.P.A.M.', 'Stocking', 42, 'Jack', 'Jill']
>>> things[2:4]
[2, 'S.P.A.M.']
>>> things[4:7]
['Stocking', 42, 'Jack']
```

```
>>> things[1:5]
['Fred', 2, 'S.P.A.M.', 'Stocking']
```

Slicing is used to return part of a list. The slicing operator is in the form `things[first_index:last_index]`. Slicing cuts the list before the `first_index` and before the `last_index` and returns the parts in between. You can use both types of indexing:

```
>>> things[-4:-2]
['Stocking', 42]
>>> things[-4]
'Stocking'
>>> things[-4:6]
['Stocking', 42]
```

Another trick with slicing is the unspecified index. If the first index is not specified the beginning of the list is assumed. If the last index is not specified the whole rest of the list is assumed. Here are some examples:

```
>>> things[:2]
[0, 'Fred']
>>> things[-2:]
['Jack', 'Jill']
>>> things[:3]
[0, 'Fred', 2]
>>> things[:-5]
[0, 'Fred', 2]
```

Here is a (HTML inspired) program example (copy and paste in the poem definition if you want):

```
poem = ["<B>", "Jack", "and", "Jill", "</B>", "went", "up", "the",
        "hill", "to", "<B>", "fetch", "a", "pail", "of", "</B>",
        "water.", "Jack", "fell", "<B>", "down", "and", "broke",
        "</B>", "his", "crown", "and", "<B>", "Jill", "came",
        "</B>", "tumbling", "after"]

def get_bolds(text):
    true = 1
    false = 0
    ## is_bold tells whether or not we are currently looking at
    ## a bold section of text.
    is_bold = false
    ## start_block is the index of the start of either an unbolded
    ## segment of text or a bolded segment.
    start_block = 0
    for index in range(len(text)):
        ## Handle a starting of bold text
        if text[index] == "<B>":
            if is_bold:
                print("Error: Extra Bold")
            ## print "Not Bold:", text[start_block:index]
            is_bold = true
            start_block = index + 1
        ## Handle end of bold text
        ## Remember that the last number in a slice is the index
        ## after the last index used.
        if text[index] == "</B>":
            if not is_bold:
                print("Error: Extra Close Bold")
            print("Bold [", start_block, ":", index, "]", text[start_block:index])
            is_bold = false
            start_block = index + 1

get_bolds(poem)
```

with the output being:

```
Bold [ 1 : 4 ] ['Jack', 'and', 'Jill']
```

```
Bold [ 11 : 15 ] ['fetch', 'a', 'pail', 'of']
Bold [ 20 : 23 ] ['down', 'and', 'broke']
Bold [ 28 : 30 ] ['Jill', 'came']
```

The `get_bold()` function takes in a list that is broken into words and tokens. The tokens that it looks for are `<B>` which starts the bold text and `</B>` which ends bold text. The function `get_bold()` goes through and searches for the start and end tokens.

The next feature of lists is copying them. If you try something simple like:

```
>>> a = [1, 2, 3]
>>> b = a
>>> print(b)
[1, 2, 3]
>>> b[1] = 10
>>> print(b)
[1, 10, 3]
>>> print(a)
[1, 10, 3]
```

This probably looks surprising since a modification to `b` resulted in `a` being changed as well. What happened is that the statement `b = a` makes `b` a *reference* to `a`. This means that `b` can be thought of as another name for `a`. Hence any modification to `b` changes `a` as well. However some assignments don't create two names for one list:

```
>>> a = [1, 2, 3]
>>> b = a * 2
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
>>> a[1] = 10
>>> print(a)
[1, 10, 3]
>>> print(b)
[1, 2, 3, 1, 2, 3]
```

In this case `b` is not a reference to `a` since the expression `a * 2` creates a new list. Then the statement `b = a * 2` gives `b` a reference to `a * 2` rather than a reference to `a`. All assignment operations create a reference. When you pass a list as an argument to a function you create a reference as well. Most of the time you don't have to worry about creating references rather than copies. However when you need to make modifications to one list without changing another name of the list you have to make sure that you have actually created a copy.

There are several ways to make a copy of a list. The simplest that works most of the time is the slice operator since it always makes a new list even if it is a slice of a whole list:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> b[1] = 10
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 10, 3]
```

Taking the slice `[:]` creates a new copy of the list. However it only copies the outer list. Any sublist inside is still a references to the sublist in the original list. Therefore, when the list contains lists, the inner lists have to be copied as well. You could do that manually but Python already contains a module to do it. You use the `deepcopy` function of the `copy` module:

```
>>> import copy
>>> a = [[1, 2, 3], [4, 5, 6]]
>>> b = a[:]
>>> c = copy.deepcopy(a)
>>> b[0][1] = 10
>>> c[1][1] = 12
>>> print(a)
[[1, 10, 3], [4, 5, 6]]
>>> print(b)
[[1, 10, 3], [4, 5, 6]]
>>> print(c)
[[1, 2, 3], [4, 12, 6]]
```

First of all notice that `a` is a list of lists. Then notice that when `b[0][1] = 10` is run both `a` and `b` are changed, but `c` is not. This happens because the inner arrays are still references when the slice operator is used. However with `deepcopy` `c` was fully copied.

So, should I worry about references every time I use a function or =? The good news is that you only have to worry about references when using dictionaries and lists. Numbers and strings create references when assigned but every operation on numbers and strings that modifies them creates a new copy so you can never modify them unexpectedly. You do have to think about references when you are modifying a list or a dictionary.

By now you are probably wondering why are references used at all? The basic reason is speed. It is much faster to make a reference to a thousand element list than to copy all the elements. The other reason is that it allows you to have a function to modify the inputed list or dictionary. Just remember about references if you ever have some weird problem with data being changed when it shouldn't be.

---

16. Revenge of the Strings

And now presenting a cool trick that can be done with strings:

```
def shout(string):
    for character in string:
        print("Gimme a " + character)
        print("'" + character + "'")

shout("Lose")

def middle(string):
    print("The middle character is:", string[len(string) // 2])

middle("abcdefg")
middle("The Python Programming Language")
middle("Atlanta")
```

And the output is:

```
Gimme a L
'L'
Gimme a o
'o'
Gimme a s
's'
Gimme a e
'e'
The middle character is: d
The middle character is: r
```

```
The middle character is: a
```

What these programs demonstrate is that strings are similar to lists in several ways. The `shout()` function shows that `for` loops can be used with strings just as they can be used with lists. The `middle` procedure shows that that strings can also use the `len()` function and array indexes and slices. Most list features work on strings as well.

The next feature demonstrates some string specific features:

```
def to_upper(string):
    ## Converts a string to upper case
    upper_case = ""
    for character in string:
        if 'a' <= character <= 'z':
            location = ord(character) - ord('a')
            new_ascii = location + ord('A')
            character = chr(new_ascii)
        upper_case = upper_case + character
    return upper_case

print(to_upper("This is Text"))
```

with the output being:

```
THIS IS TEXT
```

This works because the computer represents the characters of a string as numbers from 0 to 1,114,111. For example 'A' is 65, 'B' is 66 and א is 1488. The values are the unicode value. Python has a function called `ord()` (short for ordinal) that returns a character as a number. There is also a corresponding function called `chr()` that converts a number into a character. With this in mind the program should start to be clear. The first detail is the line: `if 'a' <= character <= 'z'`: which checks to see if a letter is lower case. If it is then the next lines are used. First it is converted into a location so that a = 0, b = 1, c = 2 and so on with the line: `location = ord(character) - ord('a')`. Next the new value is found with `new_ascii = location + ord('A')`. This value is converted back to a character that is now upper case. Note that if you really need the upper case of a letter, you should use `u=var.upper()` which will work with other languages as well.

Now for some interactive typing exercise:

```
>>> # Integer to String
>>> 2
2
>>> repr(2)
'2'
>>> -123
-123
>>> repr(-123)
'-123'
>>> # String to Integer
>>> "23"
'23'
>>> int("23")
23
>>> "23" * 2
'2323'
>>> int("23") * 2
46
>>> # Float to String
>>> 1.23
1.23
>>> repr(1.23)
'1.23'
```

```
>>> # Float to Integer
>>> 1.23
1.23
>>> int(1.23)
1
>>> int(-1.23)
-1
>>> # String to Float
>>> float("1.23")
1.23
>>> "1.23"
'1.23'
>>> float("123")
123.0
```

If you haven't guessed already the function `repr()` can convert an integer to a string and the function `int()` can convert a string to an integer. The function `float()` can convert a string to a float. The `repr()` function returns a printable representation of something. Here are some examples of this:

```
>>> repr(1)
'1'
>>> repr(234.14)
'234.14'
>>> repr([4, 42, 10])
'[4, 42, 10]'
```

The `int()` function tries to convert a string (or a float) into an integer. There is also a similar function called `float()` that will convert a integer or a string into a float. Another function that Python has is the `eval()` function. The `eval()` function takes a string and returns data of the type that python thinks it found. For example:

```
>>> v = eval('123')
>>> print(v, type(v))
123 <type 'int'>
>>> v = eval('645.123')
>>> print(v, type(v))
645.123 <type 'float'>
>>> v = eval('[1, 2, 3]')
>>> print(v, type(v))
[1, 2, 3] <type 'list'>
```

If you use the `eval()` function you should check that it returns the type that you expect.

One useful string function is the `split()` method. Here's an example:

```
>>> "This is a bunch of words".split()
['This', 'is', 'a', 'bunch', 'of', 'words']
>>> text = "First batch, second batch, third, fourth"
>>> text.split(",")
['First batch', ' second batch', ' third', ' fourth']
```

Notice how `split()` converts a string into a list of strings. The string is split by whitespace by default or by the optional argument (in this case a comma). You can also add another argument that tells `split()` how many times the separator will be used to split the text. For example:

```
>>> list = text.split(",")
>>> len(list)
4
>>> list[-1]
' fourth'
>>> list = text.split(",", 2)
>>> len(list)
```

```
3
>>> list[-1]
' third, fourth'
```

Slicing strings (and lists)

Strings can be cut into pieces — in the same way as it was shown for lists in the previous chapter — by using the *slicing* "operator" `[]`. The slicing operator works in the same way as before: text[first_index:last_index] (in very rare cases there can be another colon and a third argument, as in the example shown below).

In order not to get confused by the index numbers, it is easiest to see them as *clipping places*, possibilities to cut a string into parts. Here is an example, which shows the clipping places (in yellow) and their index numbers (red and blue) for a simple text string:

```
          0   1   2   ...  -2  -1
          ↓   ↓   ↓   ↓   ↓   ↓   ↓
text = "   S   T   R   I   N   G   "
          ↑                       ↑
          [:                      :]
```

Note that the red indexes are counted from the beginning of the string and the blue ones from the end of the string backwards. (Note that there is no blue -0, which could seem to be logical at the end of the string. Because `-0 == 0`, -0 means "beginning of the string" as well.) Now we are ready to use the indexes for slicing operations:

text[1:4]	→	"TRI"
text[:5]	→	"STRIN"
text[:-1]	→	"STRIN"
text[-4:]	→	"RING"
text[2]	→	"R"
text[:]	→	"STRING"
text[::-1]	→	"GNIRTS"

`text[1:4]` gives us all of the `text` string between clipping places 1 and 4, "`TRI`". If you omit one of the [first_index:last_index] arguments, you get the beginning or end of the string as default: `text[:5]` gives "`STRIN`". For both `first_index` and `last_index` we can use both the red and the blue numbering schema: `text[:-1]` gives the same as `text[:5]`, because the index -1 is at the same place as 5 in this case. If we do not use an argument containing a colon, the number is treated in a different way: `text[2]` gives us one character following the second clipping point, "`R`". The special slicing operation `text[:]` means "from the beginning to the end" and produces a copy of the entire string (or list, as shown in the previous chapter).

Last but not least, the slicing operation can have a second colon and a third argument, which is interpreted as the "step size": `text[::-1]` is `text` from beginning to the end, with a step size of -1. -1 means "every character, but in the other direction". "`STRING`" backwards is "`GNIRTS`" (test a step length of 2, if you have not got the point here).

All these slicing operations work with lists as well. In that sense strings are just a special case of lists, where the list elements are single characters. Just remember the concept of *clipping places*, and the indexes for slicing things will get a lot less confusing.

Examples

```
# This program requires an excellent understanding of decimal numbers.
def to_string(in_int):
    """Converts an integer to a string"""
    out_str = ""
    prefix = ""
    if in_int < 0:
        prefix = "-"
        in_int = -in_int
    while in_int // 10 != 0:
        out_str = str(in_int % 10) + out_str
        in_int = in_int // 10
    out_str = str(in_int % 10) + out_str
    return prefix + out_str

def to_int(in_str):
    """Converts a string to an integer"""
    out_num = 0
    if in_str[0] == "-":
        multiplier = -1
        in_str = in_str[1:]
    else:
        multiplier = 1
    for c in in_str:
        out_num = out_num * 10 + int(c)
    return out_num * multiplier

print(to_string(2))
print(to_string(23445))
print(to_string(-23445))
print(to_int("14234"))
print(to_int("12345"))
print(to_int("-3512"))
```

The output is:

```
2
23445
-23445
14234
12345
-3512
```

---

17. File IO

File I/O

Here is a simple example of file I/O (input/output):

```
# Write a file
with open("test.txt", "wt") as out_file:
    out_file.write("This Text is going to out file\nLook at it and see!")

# Read a file
with open("test.txt", "rt") as in_file:
    text = in_file.read()

print(text)
```

The output and the contents of the file `test.txt` are:

```
This Text is going to out file
Look at it and see!
```

Notice that it wrote a file called `test.txt` in the directory that you ran the program from. The `\n` in the string tells Python to put a *n*ewline where it is.

A overview of file I/O is:

- Get a file object with the `open` function
- Read or write to the file object (depending on how it was opened)
- If you did not use `with` to open the file, you'd have to close it manually

The first step is to get a file object. The way to do this is to use the `open` function. The format is `file_object = open(filename, mode)` where `file_object` is the variable to put the file object, `filename` is a string with the filename, and `mode` is `"rt"` to *r*ead a file as *t*ext or `"wt"` to *w*rite a file as *t*ext (and a few others we will skip here). Next the file objects functions can be called. The two most common functions are `read` and `write`. The `write` function adds a string to the end of the file. The `read` function reads the next thing in the file and returns it as a string. If no argument is given it will return the whole file (as done in the example).

Now here is a new version of the phone numbers program that we made earlier:

```
def print_numbers(numbers):
    print("Telephone Numbers:")
    for k, v in numbers.items():
        print("Name:", k, "\tNumber:", v)
    print()

def add_number(numbers, name, number):
    numbers[name] = number

def lookup_number(numbers, name):
    if name in numbers:
        return "The number is " + numbers[name]
    else:
        return name + " was not found"

def remove_number(numbers, name):
    if name in numbers:
        del numbers[name]
    else:
        print(name," was not found")

def load_numbers(numbers, filename):
    in_file = open(filename, "rt")
    while True:
        in_line = in_file.readline()
        if not in_line:
            break
        in_line = in_line[:-1]
        name, number = in_line.split(",")
        numbers[name] = number
    in_file.close()

def save_numbers(numbers, filename):
    out_file = open(filename, "wt")
    for k, v in numbers.items():
        out_file.write(k + "," + v + "\n")
    out_file.close()

def print_menu():
    print('1. Print Phone Numbers')
    print('2. Add a Phone Number')
    print('3. Remove a Phone Number')
    print('4. Lookup a Phone Number')
    print('5. Load numbers')
    print('6. Save numbers')
    print('7. Quit')
    print()

phone_list = {}
menu_choice = 0
```

```
print_menu()
while True:
    menu_choice = int(input("Type in a number (1-7): "))
    if menu_choice == 1:
        print_numbers(phone_list)
    elif menu_choice == 2:
        print("Add Name and Number")
        name = input("Name: ")
        phone = input("Number: ")
        add_number(phone_list, name, phone)
    elif menu_choice == 3:
        print("Remove Name and Number")
        name = input("Name: ")
        remove_number(phone_list, name)
    elif menu_choice == 4:
        print("Lookup Number")
        name = input("Name: ")
        print(lookup_number(phone_list, name))
    elif menu_choice == 5:
        filename = input("Filename to load: ")
        load_numbers(phone_list, filename)
    elif menu_choice == 6:
        filename = input("Filename to save: ")
        save_numbers(phone_list, filename)
    elif menu_choice == 7:
        break
    else:
        print_menu()

print("Goodbye")
```

Notice that it now includes saving and loading files. Here is some output of my running it twice:

```
1. Print Phone Numbers
2. Add a Phone Number
3. Remove a Phone Number
4. Lookup a Phone Number
5. Load numbers
6. Save numbers
7. Quit

Type in a number (1-7): 2
Add Name and Number
Name: Jill
Number: 1234
Type in a number (1-7): 2
Add Name and Number
Name: Fred
Number: 4321
Type in a number (1-7): 1
Telephone Numbers:
Name: Jill     Number: 1234
Name: Fred     Number: 4321

Type in a number (1-7): 6
Filename to save: numbers.txt
Type in a number (1-7): 7
Goodbye
```

```
1. Print Phone Numbers
2. Add a Phone Number
3. Remove a Phone Number
4. Lookup a Phone Number
5. Load numbers
6. Save numbers
7. Quit

Type in a number (1-7): 5
Filename to load: numbers.txt
Type in a number (1-7): 1
Telephone Numbers:
Name: Jill     Number: 1234
Name: Fred     Number: 4321

Type in a number (1-7): 7
```

```
Goodbye
```

The new portions of this program are:

```
def load_numbers(numbers, filename):
    in_file = open(filename, "rt")
    while True:
        in_line = in_file.readline()
        if not in_line:
            break
        in_line = in_line[:-1]
        name, number = in_line.split(",")
        numbers[name] = number
    in_file.close()

def save_numbers(numbers, filename):
    out_file = open(filename, "wt")
    for k, v in numbers.items():
        out_file.write(k + "," + v + "\n")
    out_file.close()
```

First we will look at the save portion of the program. First it creates a file object with the command `open(filename, "wt")`. Next it goes through and creates a line for each of the phone numbers with the command `out_file.write(k + "," + v + "\n")`. This writes out a line that contains the name, a comma, the number and follows it by a newline.

The loading portion is a little more complicated. It starts by getting a file object. Then it uses a `while True:` loop to keep looping until a `break` statement is encountered. Next it gets a line with the line `in_line = in_file.readline()`. The `readline` function will return a empty string when the end of the file is reached. The `if` statement checks for this and `breaks` out of the `while` loop when that happens. Of course if the `readline` function did not return the newline at the end of the line there would be no way to tell if an empty string was an empty line or the end of the file so the newline is left in what `readline` returns. Hence we have to get rid of the newline. The line `in_line = in_line[:-1]` does this for us by dropping the last character. Next the line `name, number = in_line.split(",")` splits the line at the comma into a name and a number. This is then added to the `numbers` dictionary.

Advanced use of .txt files

You might be saying to yourself, "Well I know how to read and write to a textfile, but what if I want to print the file without opening out another program?"

There are a few different ways to accomplish this. The easiest way does open another program, but everything is taken care of in the Python code, and doesn't require the user to specify a file to be printed. This method involves invoking the subprocess of another program.

Remember the file we wrote output to in the above program? Let's use that file. Keep in mind, in order to prevent some errors, this program uses concepts from the Next chapter. Please feel free to revisit this example after the next chapter.

```
import subprocess
def main():
    try:
        print("This small program invokes the print function in the Notepad application")
        #Lets print the file we created in the program above
        subprocess.call(['notepad','/p','numbers.txt'])
    except WindowsError:
        print("The called subprocess does not exist, or cannot be called.")

main()
```

The `subprocess.call` takes three arguments. The first argument in the context of this example, should be the name of the program which you would like to invoke the printing subprocess from. The second argument should be the specific subprocess within that program. For simplicity, just understand that in this program, `'/p'` is the subprocess used to access your printer through the specified application. The last argument should be the name of the file you want to send to the printing subprocess. In this case, it is the same file used earlier in this chapter.

Exercises

Now modify the grades program from section Dictionaries so that is uses file I/O to keep a record of the students.

Solution

Now modify the grades program from section Dictionaries so that is uses file I/O to keep a record of the students.

```
assignments = ['hw ch 1', 'hw ch 2', 'quiz   ', 'hw ch 3', 'test']
students = { }

def load_grades(gradesfile):
    inputfile = open(gradesfile, "r")
    grades = [ ]
    while True:
        student_and_grade = inputfile.readline()
        student_and_grade = student_and_grade[:-1]
        if not student_and_grade:
            break
        else:
            studentname, studentgrades = student_and_grade.split(",")
            studentgrades = studentgrades.split(" ")
            students[studentname] = studentgrades
    inputfile.close()
    print("Grades loaded.")

def save_grades(gradesfile):
    outputfile = open(gradesfile, "w")
    for k, v in students.items():
        outputfile.write(k + ",")
        for x in v:
            outputfile.write(str(x) + " ")
        outputfile.write("\n")
    outputfile.close()
    print("Grades saved.")

def print_menu():
    print("1. Add student")
    print("2. Remove student")
    print("3. Load grades")
    print("4. Record grade")
    print("5. Print grades")
    print("6. Save grades")
    print("7. Print Menu")
    print("9. Quit")

def print_all_grades():
    if students:
        keys = sorted(students.keys())
        print('\t', end=' ')
        for x in assignments:
            print(x, '\t', end=' ')
        print()
        for x in keys:
            print(x, '\t', end=' ')
            grades = students[x]
            print_grades(grades)
    else:
        print("There are no grades to print.")

def print_grades(grades):
```

```
    for x in grades:
        print(x, '\t', end=' ')
    print()

print_menu()
menu_choice = 0
while menu_choice != 9:
    print()
    menu_choice = int(input("Menu Choice: "))
    if menu_choice == 1:
        name = input("Student to add: ")
        students[name] = [0] * len(assignments)
    elif menu_choice == 2:
        name = input("Student to remove: ")
        if name in students:
            del students[name]
        else:
            print("Student:", name, "not found")
    elif menu_choice == 3:
        gradesfile = input("Load grades from which file? ")
        load_grades(gradesfile)
    elif menu_choice == 4:
        print("Record Grade")
        name = input("Student: ")
        if name in students:
            grades = students[name]
            print("Type in the number of the grade to record")
            print("Type a 0 (zero) to exit")
            for i,x in enumerate(assignments):
                print(i + 1, x, '\t', end=' ')
            print()
            print_grades(grades)
            which = 1234
            while which != -1:
                which = int(input("Change which Grade: "))
                which -= 1
                if 0 <= which < len(grades):
                    grade = input("Grade: ") # Change from float(input()) to input() to avoid an error w
                    grades[which] = grade
                elif which != -1:
                    print("Invalid Grade Number")
        else:
            print("Student not found")
    elif menu_choice == 5:
        print_all_grades()
    elif menu_choice == 6:
        gradesfile = input("Save grades to which file? ")
        save_grades(gradesfile)
    elif menu_choice != 9:
        print_menu()
```

---

18. Dealing with the imperfect

...or how to handle errors

closing files with with

We use the "with" statement to open and close files.[1][2]

```
with open("in_test.txt", "rt") as in_file:
    with open("out_test.txt", "wt") as out_file:
        text = in_file.read()
        data = parse(text)
        results = encode(data)
        out_file.write(results)
    print( "All done." )
```

If some sort of error happens anywhere in this code (one of the files is inaccessible, the parse() function chokes on corrupt data, etc.) the "with" statements guarantee that all the files will eventually be properly closed. Closing a file just means that the file is "cleaned up" and "released" by our program so that it can be used in another program.

catching errors with try

So you now have the perfect program, it runs flawlessly, except for one detail, it will crash on invalid user input. Have no fear, for Python has a special control structure for you. It's called `try` and it tries to do something. Here is an example of a program with a problem:

```
print("Type Control C or -1 to exit")
number = 1
while number != -1:
   number = int(input("Enter a number: "))
   print("You entered:", number)
```

Notice how when you enter `@#&` it outputs something like:

```
Traceback (most recent call last):
 File "try_less.py", line 4, in <module>
   number = int(input("Enter a number: "))
ValueError: invalid literal for int() with base 10: '\\@#&'
```

As you can see the `int()` function is unhappy with the number `@#&` (as well it should be). The last line shows what the problem is; Python found a `ValueError`. How can our program deal with this? What we do is first: put the place where errors may occur in a `try` block, and second: tell Python how we want `ValueErrors` handled. The following program does this:

```
print("Type Control C or -1 to exit")
number = 1
while number != -1:
    try:
        number = int(input("Enter a number: "))
        print("You entered:", number)
    except ValueError:
        print("That was not a number.")
```

Now when we run the new program and give it `@#&` it tells us "That was not a number." and continues with what it was doing before.

When your program keeps having some error that you know how to handle, put code in a `try` block, and put the way to handle the error in the `except` block.

Exercises

Update at least the phone numbers program (in section Dictionaries) so it doesn't crash if a user doesn't enter any data at the menu.

---

19. The End

So here we are at the end, or maybe the beginning. This tutorial is on Wikibooks, so feel free to make improvements to it. If you want to learn more about Python, The Python Tutorial (http://docs.python.org

/3/tutorial/index.html) by Guido van Rossum (http://www.python.org/~guido/) has more topics that you can learn about. If you have been following this tutorial, you should be able to understand a fair amount of it. The Python Programming wikibook can be worth looking at, too. Here are few other books which cover Python 3:

- A Byte of Python by Swaroop C H (http://www.swaroopch.com/notes/Python)
- Hands-on Python Tutorial by Dr. Andrew N. Harrington (http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html)
- Subject:Python programming language lists other Wikibooks related to Python.

Hopefully this book covers everything you have needed to get started programming. Thanks to everyone who has sent me emails about it. I enjoyed reading them, even when I have not always been the best replier.

Happy programming, may it change your life and the world.

---

20. FAQ

How do I make a GUI in Python?
: You can use one of these library: TKinter (https://docs.python.org/3.5/library/tkinter.html), PyQt (https://riverbankcomputing.com/software/pyqt/intro), PyGobject (https://wiki.gnome.org/Projects/PyGObject). For really simple graphics, you can use the turtle graphics mode `import turtle`

How do I make a game in Python?
: The best method is probably to use PyGame at http://pygame.org/

How do I make an executable from a Python program?
: Short answer: Python is an interepreted language so that is impossible. Long answer is that something similar to an executable can be created by taking the Python interpreter and the file and joining them together and distributing that. For more on that problem see http://www.python.org/doc/faq/programming/#how-can-i-create-a-stand-alone-binary-from-a-python-script

(IFAQ) Why do you use first person in this tutorial?
: Once upon a time in a different millenia, (1999 to be exact), an earlier version was written entirely by Josh Cogliati, and it was up on his webpage http://www.honors.montana.edu/~jjc/easytut and it was good. Then the server rupert, like all good things than have a beginning came to an end, and Josh moved it to Wikibooks, but the first person writing stuck. If someone really wants to change it, I will not revert it, but I don't see much point.

My question is not answered.
: Ask on the discussion page or add it to this FAQ, or email one of the Authors.

For other FAQs, you may want to see the Python 2.6 version of this page Non-Programmer's Tutorial for Python 2.6/FAQ, or the Python FAQ (https://docs.python.org/3.5/faq/).

---

1. "The 'with' statement" (http://docs.python.org/3.4/reference/compound_stmts.html#the-with-statement)
2. 'The Python "with" Statement by Example' (http://preshing.com/20110920/the-python-with-statement-by-example/)
