An introduction to Python through practical examples

Fletcher Heisler

0) Introduction.....5

0.1) Why Python?.....5
0.2) Why this book?.....6
0.3) How to use this book.....7
0.4) License.....8

1) Getting Started.....9

1.1) Download Python.....9
1.2) Open IDLE.....9
1.3) Write a Python script.....10
1.4) Screw things up.....12
1.5) Store a variable.....14

Interlude: Leave yourself helpful notes.....16

2) Fundamentals: Strings and Methods.....18

2.1) Learn to speak in Python.....18
2.2) Mess around with your words.....19
2.3) Use objects and methods.....23
Assignment 2.3: Pick apart your user's input.....26

3) Fundamentals: Working with Strings.....27

3.1) Mix and match different objects.....27
3.2) Streamline your print statements.....29
3.3) Find a string in a string.....31
Assignment 3.3: Turn your user into a l33t h4x0r.....33

4) Fundamentals: Functions and Loops.....34

4.1) Do futuristic arithmetic.....34
Assignment 4.1: Perform calculations on user input.....36
4.2) Create your own functions.....37
Assignment 4.2: Convert temperatures.....40
4.3) Run in circles.....40
Assignment 4.3: Track your investments.....44

Interlude: Debug your code.....46

5) Fundamentals: Conditional logic.....51

5.1) Compare values.....51
5.2) Add some logic.....53
5.3) Control the flow of your program.....58
Assignment 5.3: Find the factors of a number.....61
5.4) Break out of the pattern.....62
5.5) Recover from errors.....65
5.6) Simulate events and calculate probabilities.....68
Assignment 5.6.1: Simulate an election.....70
Assignment 5.6.2: Simulate a coin toss experiment.....71

6) Fundamentals: Lists and Dictionaries.....72

6.1) Make and update lists.....72
Assignment 6.1: Wax poetic.....77
6.2) Make permanent lists.....78
6.3) Store relationships in dictionaries.....80

7) File Input and Output.....86

7.1) Read and write simple files.....86
7.2) Use more complicated folder structures.....91
Assignment 7.2: Use pattern matching to delete files.....97
7.3) Read and write CSV data.....98
Assignment 7.3: Create a high scores list from CSV data.....102

Interlude: Install packages.....103

8) Interact with PDF files.....107

8.1) Read and write PDFs.....107
8.2) Manipulate PDF files.....111
Assignment 8.2: Add a cover sheet to a PDF file.....115

9) SQL database connections.....116

9.1) Communicate with databases using SQLite.....116
9.2) Use other SQL variants.....122

10) Interacting with the web.....124

10.1) Scrape and parse text from websites.....124
10.2) Use an HTML parser to scrape websites.....131
10.3) Interact with HTML forms.....135
10.4) Interact with websites in real-time.....142

11) Scientific computing and graphing.....145

11.1) Use NumPy for matrix manipulation.....145
11.2) Use matplotlib for plotting graphs.....151
Assignment 11.2: Plot a graph from CSV data.....164

12) Graphical User Interfaces.....165

12.1) Add GUI elements with EasyGUI.....165
Assignment 12.1: Use GUI elements to help a user modify files.....172

12.2) Create GUI applications with Tkinter.....172
Assignment 12.2: Return of the poet.....187

13) Web applications.....188

13.1) Create a simple web application.....188
13.2) Create an interactive web application.....196
Assignment 13.2: The poet gains a web presence.....201

13.3) Put your web application online.....202

Final Thoughts.....204

Acknowledgements.....205

0) Introduction

Whether you're new to programming or a professional code monkey looking to dive into a new language, this book will teach you all of the *practical* Python that you need to get started on projects on your own.

Real Python emphasizes real-world programming techniques, which are illustrated through interesting, useful examples. No matter what your ultimate goals may be, if you work with computer at all, you will soon be finding endless ways to improve your life by automating tasks and solving problems through Python programs that you create.

0.1) Why Python?

Python is open-source freeware, meaning you can download it for free and use it for any purpose. It also has a great support community that has built a number of additional free tools. Need to work with PDF documents in Python? There's a free package for that. Want to collect data from webpages? No need to start from scratch!

Python was built to be easier to use than other programming languages. It's usually *much* easier to read Python code and MUCH faster to write code in Python than in other languages.

For instance, here's some simple code written in C, another commonly used programming language:

```
#include <stdio.h>
int main ()
{
  printf ("Hello, world\n");
}
```

All the program does is print “Hello, world” on the screen. That was a lot of work to print one phrase! Here's the same code in Python:

```
print "Hello, world"
```

Simple, right? Easy, faster, more readable.

At the same time, Python has all the functionality of other languages and more. You might be surprised how many professional products are built on Python code: Gmail, Google Maps, YouTube, reddit, Spotify, turntable.fm, Yahoo! Groups, and the list goes on… And if it's powerful enough for both NASA and the NSA, it's good enough for us.

0.2) Why this book?

There are *tons* of books and tutorials out there for learning Python already. However, most of the resources out there generally have two main problems:

1) They aren't practical.
2) They aren't interesting.

Most books are so preoccupied with covering *every* last possible variation of *every* command that it's easy to get lost in the details. In the end, most of them end up looking more like the Python documentation pages. This is great as reference material, but it's a horrible way to learn a programming language. Not only do you spend most of your time learning things you'll never use, but it *isn't any fun!*

This book is built on the 80/20 principle. We will cover the commands and techniques used in the *vast* majority of cases and focus on how to program real-world solutions to problems that ordinary people *actually* want to solve.

This way, I guarantee that you will:

- Learn useful techniques much faster
- Spend less time struggling with unimportant complications
- Find more practical uses for Python in your own life
- Have more fun in the process!

If you want to become a serious, professional Python programmer, this book won't be enough by itself – but it will *still* be the best starting point. Once you've mastered the material in this book, you will have gained a strong enough foundation that venturing

out into more advanced territory on your own will be a breeze.

So dive in! Learn to program in a widely used, free language that can do more than you ever thought was possible.

0.3) How to use this book

For the most part, you should approach the topics in the first half of this book in the same order as they are presented. This is less true of the second half, which covers a number of mostly non-overlapping topics, although the chapters are generally increasing in difficulty throughout. If you are a more experienced programmer, then you may find yourself heading toward the back of the book right away – but don't neglect getting a strong foundation in the basics first!

Each chapter section is followed by review exercises to help you make sure that you've mastered all the topics covered. There are also a number of assignments, which are more involved and usually require you to tie together a number of different concepts from previous chapters. The practice files that accompany this course also include solution scripts to the assignments as well as some of the trickier exercises – but to get the most out of them, you should try your best to solve the assignment problems on your own before looking at the example solutions.

If you get stuck, you can always log in at RealPython.com and ask for help on the members' forum; it's likely that someone else has already experienced the same difficulty that you're encountering and might be able to guide you along.

This book does move quickly, however, so if you're *completely* new to programming, you may want to supplement the first few chapters with additional practice. I highly recommend working through the beginning Python lessons available for free at the Codecademy site *while* you make your way through the beginning of this material as the best way to make sure that you have all the basics down.

Finally, if you have any questions or feedback about the course, you're always welcome to contact me directly.

0.4) License

1) Getting Started

1.1) Download Python

Before we can do anything, you have to download Python. Even if you already have Python on your computer, make sure that you have the correct version: **2.7.3** is the version used in this book and by most of the rest of the world.

! There's a newer version, Python 3.2, but it can't run code that was created with previous versions of Python (including a lot of useful and important packages that haven't been updated). As a result, Python 3 still hasn't caught on yet. Since most of the code you'll see elsewhere will be from Python 2.7, you should learn that first. The two versions are still *very* similar, and it will take you very little time to get used to the minor changes in Python 3 after you've mastered Python 2.

Mac users: You already have a version of Python installed by default, but it's *not quite the same* as the standard installation. You should still download Python 2.7.3 as directed below. Otherwise, you might run into problems later when trying to install some additional functionality in Python or running code that involves graphics windows.

Linux users: You might already have Python 2.7.3 installed by default. Open your Terminal application and type "`python --version`" to find out. If you have 2.7.1 or 2.7.2, you should go ahead and update to the latest version.

If you need to, go to http://www.python.org/download/ to download **Python 2.7.3** for your operating system and install the program.

1.2) Open IDLE

We'll be using IDLE (**I**nteractive **D**eve**L**opment **E**nvironment) to write our Python code. IDLE is a simple editing program that comes automatically installed with

Python on Windows and Mac, and it will make our lives *much* easier while we're coding. You could write Python scripts in any program from a basic text editor to a very complex development environment (and many professional coders use more advanced setups), but IDLE is simple to use and will easily provide all the functionality we need.

Windows: Go to your start menu and click on “IDLE (Python GUI)” from the “Python 2.7” program folder to open IDLE. You can also type “IDLE” into the search bar.
OS X: Go to your Applications folder and click on “IDLE” from the “Python 2.7” folder to start running IDLE. Alternatively, you can type “`IDLE`” (without quotes) into your Terminal window to launch IDLE.
Linux: I recommend that you install IDLE to follow along with this course. You could use Vim or Emacs, but they will not have the same built-in debugging features. To install IDLE with admin privileges: On Ubuntu/Debian, type: `sudo apt-get install idle` On Fedora/Red Hat/RHEL/CentOS, type: `sudo yum install python-tools` On SUSE, you can search for IDLE via “install software” through YaST.

Opening IDLE, you will see a brief description of Python, followed by a prompt:

```
>>>
```

We're ready to program!

1.3) Write a Python script

The window we have open at the moment is IDLE's *interactive window*; usually this window will just show us results when we run programs that we've written, but we can also enter Python code into this window directly. Go ahead and try typing some basic math into the interactive window at the prompt - when you hit enter, it should evaluate your calculation, display the result and prompt you for more input:

```
>>> 1+1
2
>>>
```

Let's try out some actual code. The standard program to display “Hello, world” on the screen is *just that simple* in Python. Tell the interactive window to print the phrase by using the `print` command like so:

```
>>> print "Hello, world"
Hello, world
>>>
```

If you want to get to previous lines you've typed into the interactive window without typing them out again or copying and pasting, you can use the pair of shortcut keys ALT+P (or on a Mac, CTRL+P). Each time you hit ALT+P, IDLE will fill in the previous line of code for you. You can then type ALT+N (OS X: CTRL+N) to cycle back to the next most recent line of code.

Normally we will want to run more than one line of code at a time and save our work so that we can return to it later. To do this, we need to create a new script.

From the menu bar, choose “File → New Window” to generate a blank script. You should rearrange this window and your interactive results window so that you can see them both at the same time.

Type the same line of code as before, but put it in your new script:

```
print "Hello, world"
```

If you just copy and paste from the interactive window into the script, make sure you never include the “>>> ” part of the line. That's just the window asking for your input; it isn't part of the actual code.

In order to run this script, we need to save it first. Choose “File → Save As…”, name the file “hello_world.py” (without the quotation marks) and save it somewhere you'll be able to find it later. The “.py” extension lets IDLE know that it's a Python script.

Notice that `print` and `"Hello, world"` appear in different colors to let you know that `print` is a command and `"Hello, world"` is a string of characters. If you save the script as something other than a “`.py`” file (or if you don't include the “`.py`” extension, this coloring will disappear and everything will turn black, letting you know that the file is no longer recognized as a Python script.

Now that the script has been saved, all we have to do in order to run the program is to

select “Run → Run Module” from the script window (or hit F5), and we'll see the result appear in the interactive results window just like it did before:

```
>>>
Hello, world
>>>
```

To open and edit a program later on, just open up IDLE again and select “File → Open…”, then browse to and select the script to open it in a new script window.

Linux users: Read this overview first (especially section 2.2.2) if you want to be able to run Python scripts outside of the editor.

You might see something like the following line in the interactive window when you run or re-run a script:

```
>>> ========================== RESTART ==========================
```

This is just IDLE's way of letting you know that everything after this line is the result of the new script that you are just about to run. Otherwise, if you ran one script after another (or one script *again* after itself), it might not be clear what output belongs to which run of which script.

1.4) Screw things up

Everybody makes mistakes – especially while programming. In case you haven't made any mistakes yet, let's get a head start on that and mess something up on purpose to see what happens.

Using IDLE, there are two main types of errors you'll experience. The most common is a *syntax* error, which usually means that you've typed something incorrectly.

Let's try changing the contents of the script to:

```
print "Hello, world
```

Here we've just removed the ending quotation mark, which is of course a mistake – now

Python won't be able to tell where the string of text ends. Save your script and try running it. What happens?

…You can't run it! IDLE is smart enough to realize there's an error in your code, and it stops you from even *trying* to run the buggy program. In this case, it says: “EOL while scanning string literal.” EOL stands for “End Of Line”, meaning that Python got all the way to the end of the line and never found the end of your string of text.

IDLE even highlights the place where the error occurred using a different color and moves your cursor to the location of the error. Handy!

The other sort of error that you'll experience is the type that IDLE *can't* catch for you until your code is already running. Try changing the code in your script to:

```
print Hello, world
```

Now we've entirely removed the quotation marks from the phrase. Notice how the text changes color when we do that? IDLE is letting us know that this is no longer a string of text that we will be printing, but something else. What is our code doing now? Well, save the script and try to run it…

The interactive window will pop up with ugly red text that looks something like this:

```
>>>

Traceback (most recent call last):
  File "[path to your script]\hello world.py", line 1, in <module>
    print Hello, world
NameError: name 'Hello' is not defined
>>>
```

So what happened? Python is telling us a few things:

- An error occurred – specifically, Python calls it a `NameError`
- The error happened on `line 1` of the script
- The line that generated the error was: `print Hello, world`
- The specific error was: `name 'Hello' is not defined`

This is called a *run-time* error since it only occurs once the programming is already running. Since we didn't put quotes around `Hello, world`, Python didn't know that this

was text we wanted to print. Instead, it thought we were referring to two variables that we wanted to `print`. The first variable it tried to print was something named “Hello” - but since we hadn't defined a variable named “Hello”, our program crashed.

Review exercises:

- Write a script that IDLE won't let you run because it has a *syntax* error
- Write a script that will only crash your program once it is already running because it has a *run-time* error

1.5) Store a variable

Let's try writing a different version of the previous script. Here we'll use a variable to store our text before printing the text to the screen:

```
phrase = "Hello, world"
print phrase
```

Notice the difference in where the quotation marks go from our previous script. We are creating a variable named `phrase` and assigning it the value of the string of text `"Hello, world"`. We then print the phrase to the screen. Try saving this script and running these two lines; you should see the same output as before:

```
>>>
Hello, world
>>>
```

Notice that in our script we didn't say:

```
print "phrase"
```

Using quotes here would just print the word “phrase” instead of printing the contents of the variable named `phrase`.

! Phrases that appear in quotation marks are called *strings*. We call them strings because they're just that - strings of characters. A string is one of the most basic building blocks of any programming language, and we'll use strings a lot over the next few chapters.

We also didn't type our code like this:

```
Phrase = "Hello, world"
print phrase
```

Can you spot the difference? In this example, the first line defines the variable `Phrase` with a capital “P” at the beginning, but the second line prints out the variable `phrase`.

Since Python is case-sensitive, the variables `Phrase` and `phrase` are entirely different things. Likewise, commands start with lowercase letters; we can tell Python to `print`, but it wouldn't know how to `Print`. Keep this important distinction in mind!

When you run into trouble with the sample code, be sure to double-check that every character in your code (often including spaces) *exactly* matches the examples. Computers don't have any common sense to interpret what you *meant* to say, so being almost correct still won't get a computer to do the right thing!

Review exercises:

- Using the interactive window, display some text on the screen by using `print`
- Using the interactive window, display a string of text by saving the string to a variable, then `print`ing the contents of that variable
- Do each of the first two exercises again by first saving your code in a script and then running the script

Interlude: Leave yourself helpful notes

As you start to write more complicated scripts, you'll start to find yourself going back to parts of your code after you've written them and thinking, “what the heck was that supposed to do?”

To avoid these moments, you can leave yourself notes in your code; they don't affect the way the script runs at all, but they help to document what's supposed to be happening. These notes are referred to as *comments*, and in Python you start a comment with a pound (`#`) sign.[1] Our first script could have looked like this:

```
# This is my first script!
phrase = "Hello, world."
print phrase # this line displays "Hello, world"
```

The first line doesn't do anything, because it starts with a `#`. This tells Python to ignore the line completely because it's just a note for you.

Likewise, Python ignores the comment on the last line; it will still `print phrase`, but everything starting with the `#` is simply a comment.

Of course, you can still use a # symbol inside of a string. For instance, Python won't mistake the following for the start of a comment:

```
print "#1"
```

If you have a lot to say, you can also create comments that span over multiple lines by using a series of three single quotes (`'''`) or three double quotes (`"""`) without any spaces between them. Once you do that, *everything* after the `'''` or `"""` becomes a comment until you *close* the comment with a matching `'''` or `"""`. For instance, if you were feeling excessively verbose, our first script could have looked like this:

```
''' This is my first script.
It prints the phrase "Hello, world."
The comments are longer than the script! '''
phrase = "Hello, world."
print phrase
""" The line above displays "Hello, world" """
```

1 The `#` symbol is alternately referred to as a number sign, a hash, a crosshatch, or an octothorp. Really.

The first three lines are now all one comment, since they fall between pairs of `'''`. You can't add a multi-line comment at the end of a line of code like with the # version, which is why the last comment is on its own separate line. (We'll see why in the next chapter.)

Besides leaving yourself notes, another common use of comments is to “comment out code” while you're testing parts of a scripts to temporarily stop that part of the code from running. In other words, adding a # at the beginning of a line of code is an easy way to make sure that you don't actually use that line, even though you might want to keep it and use it later.

2) Fundamentals: Strings and Methods

2.1) Learn to speak in Python

As we've already seen, you write strings in Python by surrounding them with quotes. You can use single quotes or double quotes, as long as you're consistent for any one string. All of the following lines create string variables (called *string literals* because we've literally written out exactly how they look):

```
phrase = 'Hello, world.'
myString = "We're #1!"
stringNumber = "1234"
conversation = 'I said, "Put it over by the llama."'
```

Strings can include *any* characters - letters, numbers and symbols. You can see the benefit of using either single or double quotes in the last string example; since we used single quotes, we didn't have any trouble putting double quotes *inside* the string. (There are other ways to do this, but we'll get to those later in the chapter.)

We can also create really long strings that take up multiple lines by using three single quotes (or three double quotes), like this:

```
longString = '''This is a
string that spans across multiple lines'''

longString = """This is a new string
that spans across two lines"""
```

Here we assigned one value to the variable `longString`, then we overwrote that value with a new string literal. Try putting this in a script and then `print` the variable `longString`; you'll see that it displays the string on two separate lines. You can also see now why you can't have multi-line comments appear on the same line as actual code; Python wouldn't be able to tell the difference between these and actual string variables!

One last thing about strings: if you want to write out a really long string, but you *don't*

want it to appear on multiple lines, you can use a backslash like this when writing it out:

```
myLongString = "Here's a string that I want to write \
across multiple lines since it is long."
```

Normally Python would get to the end of the first line and get angry with you because you hadn't closed the string with a matching single quote. But because there's a backslash at the end, you can just keep writing the same string on the next line. This is different from the last example since the *actual* string isn't stored on multiple lines this time, therefore the string gets displayed on a single line without the break:

```
>>> print myLongString
Here's a string that I want to write across multiple lines since it is long.
>>>
```

! As we've already discussed, Python is case-sensitive. By convention, Python's built-in functions and methods use exclusively lower-case. Since a single variable name can't include any spaces or dashes, when programmers want to give descriptive names to variables, one way of making them easily readable is to use *camelCase* (i.e., `myLongString`), so called because of the upper-case "humps" in the middle of terms. We'll mostly stick to camelCase in this course, but another popular method is to separate words using underscores (i.e., `my_long_string`).

Review exercises:

- `print` a string that uses double quotation marks *inside* the string
- `print` a string that uses an apostrophe (single quote) *inside* the string
- `print` a string that spans across multiple lines
- `print` a one-line string that you have written out on multiple lines

2.2) Mess around with your words

Python has some built-in functionality that we can use to modify our strings or get more information about them. For instance, there's a "length" function

(abbreviated as "`len`" in Python) that can tell you the length of all sorts of things, including strings. Try typing these lines into the interactive window:

```
>>> myString = "abc"
>>> lengthOfString = len(myString)
>>> print lengthOfString
3
>>>
```

First we created a string named `myString`. Then we used the `len()` function on `myString` to calculate its length, which we store in the new variable we named `lengthOfString`. We have to give the `len()` function some *input* for its calculation, which we do by placing `myString` after it in the parentheses - you'll see more on exactly how this works later. The length of `'abc'` is just the total number of characters in it, 3, which we then `print` to the screen.

We can combine strings together as well:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magicString = string1 + string2
>>> print magicString
abracadabra
>>>
```

Or even like this, without creating any new variables:

```
>>> print "abra" + "ca" + "dabra"
abracadabra
>>>
```

In programming, when we add strings together like this, we say that we *concatenate* them.

You'll see a lot of italicized terms throughout the first few chapters of this book. Don't worry about memorizing all of them if they're unfamiliar! You don't *need* any fancy jargon to program well, but it's good to be aware of the correct terminology. Programmers tend to throw around technical terms a lot; not only does it allow for more precise communication, but it helps make simple concepts sound more impressive.

When we want to combine many strings at once, we can also use commas to separate them. This will automatically add spaces between the strings, like so:

```
>>> print "abra", "ca", "dabra"
abra ca dabra
>>>
```

Of course, the commas have to go *outside* of the quotation marks, since otherwise the commas would become part of the actual strings themselves.

Since a string is just a sequence of characters, we should be able to access each character individually as well. We can do this by using square brackets after the string, like this:

```
>>> flavor = "birthday cake"
>>> print flavor[3]
t
>>>
```

Wait, but “t” is the fourth character! Well, not in the programming world… In Python (and most other programming languages), **we start counting at 0**. So in this case, “b” is the “zeroth” character of the string “`birthday cake`”. This makes “i” the *first* character, “r” the second, and “t” the third.

If we wanted to display what we would normally tend to think of as the “first” character, we would actually need to print the 0th character:

```
>>> print flavor[0]
b
>>>
```

Be careful when you're using:
- **parentheses: `( )`**
- **square brackets: `[ ]`**
- **curly braces: `{ }`**

These *all* mean different things to Python, so you can never switch one for another. We'll see more examples of when each one is used (and we haven't seen `{}` yet), but keep in mind that they're all used differently.

The number that we assigned to each character's position is called the *index* or *subscript* number, and Python thinks of the string like this:

Character:	**b**	**i**	**r**	**t**	**h**	**d**	...
Index / Subscript #:	0	1	2	3	4	5	...

We can get a particular section out of the string as well, by using square brackets and specifying the *range* of characters that we want. We do this by putting a colon between the two subscript numbers, like so:

```
>>> flavor = "birthday cake"
>>> print flavor[0:3]
bir
>>>
```

Here we told Python to show us only the first three characters of our string, starting at the 0th character and going up *until* (but not including) the 3rd character. The number before the colon tells Python the first character we want to include, while the number after the colon says that we want to stop *just before* that character.

If we use the colon in the brackets but omit one of the numbers in a range, Python will assume that we meant to go all the way to the end of the string in that direction:

```
>>> flavor = "birthday cake"
>>> print flavor[:5]
birth
>>> print flavor[5:]
day cake
>>> print flavor[:]
birthday cake
>>>
```

The way we're using brackets after the string is referred to as *subscripting* or *indexing* since it uses the index numbers of the string's characters.

Python strings are *immutable*, meaning that they can't be changed once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
myString = "goal"
myString[0] = "f"  # this won't work!
```

Instead, we would have to create an entirely new string (although we can still give `myString` that *new* value):

```
myString = "goal"
myString = "f" + myString[1:]
```

In the first example, we were trying to change *part* of `myString` and keep the rest of it unchanged, which doesn't work. In the second example, we created a *new* string by adding two strings together, one of which was a part of `myString`; *then* we took that *new* string and completely reassigned `myString` to this new value.

Review exercises:

- Create a string and `print` its length using the `len()` function
- Create two strings, concatenate them (add them next to each other) and `print` the combination of the two strings
- Create two string variables, then `print` one of them after the other (with a space added in between) using a comma in your `print` statement
- `print` the string "`zing`" by using subscripting and index numbers on the string "`bazinga`" to specify the correct range of characters

2.3) Use objects and methods

The Python programming language is an example of *Object-Oriented Programming* (*OOP*), which means that we store our information in *objects*. In Python, a string is an example of an object.[2] Strings are very simple objects - they only hold one piece of information (their value) – but a single object can be very complex. Objects can even hold other objects inside of them. This helps to give structure and organization to our programming.

For instance, if we wanted to model a car, we would (hypothetically) create a Car object that holds lots of descriptive information about the car, called its *attributes*. It would have a color attribute, a model attribute, etc., and each of these attributes would hold one piece of descriptive information about the car. It would also include different objects like Tires, Doors, and an Engine that all have their own attributes as well.

Different objects also have different capabilities, called *methods*. For instance, our Car object might have a `drive()` method and a `park()` method. Since these methods *belong* to the car, we use them with "*dot notation*" by putting them next to the object

2 Strings are actually called `str` objects in Python, because programmers are lazy and don't want to type more than is absolutely necessary. But you'll always hear string objects referred to as just "strings."

and after a period, like this:

```
car.park()
```

Methods are followed by parentheses, because sometimes methods use *input*. For instance, if we wanted to drive the car object a distance of 50, we would place that input of 50 in the parentheses of the "drive" method:

```
car.drive(50)
```

There are certain methods that belong to string objects as well. For instance, there is a string method called `upper()` that creates an upper-case version of the string. (Likewise, there is a corresponding method `lower()` that creates a lower-case version of a string.) Let's give it a try in the interactive window:

```
>>> loudVoice = "Can you hear me yet?"
>>> print loudVoice.upper()
CAN YOU HEAR ME YET?
>>>
```

We created a string `loudVoice`, then we called its `upper()` method to return the upper-case version of the string, which we `print` to the screen.

! Methods are just functions that *belong* to objects. We already saw an example of a general-purpose function, the `len()` function, which can be used to tell us the length of many *different* types of objects, including strings. This is why we use the length function differently, by only saying:

```
len(loudVoice)
```

Meanwhile, we use dot notation to call methods that *belong* to an object, like when we call the `upper()` method that belongs to the string `loudVoice`:

```
loudVoice.upper()
```

Let's make things more interactive by introducing one more general function. We're going to get some input from the user of our program by using the function `raw_input()`. The input that we pass to this function is the text that we want it to display as a prompt; what the function actually *does* is to receive additional input from the user. Try running the following script: