# A Quick Introduction to Python

Prepared by Assil Taoufik Ksiksi

March 2014

# Contents

# Introduction

## What is Python?

"Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. The language provides constructs intended to enable clear programs on both a small and large scale.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library."[1]

### Keywords

- High-level
- Code readability and conciseness (whitespace)
- Large standard library
- Multiple paradigms
- Dynamic type system
- General purpose (scientific, web development, mobile app dev, etc.)

## The Workshop

### About Me

My name is Assil Taoufik Ksiksi, and I'm a 3rd year UAEU student pursuing a degree in Electrical Engineering. I taught myself programming 4 years ago, starting with C/C++, and have been writing Python for the past 2 years. I think of myself as an intermediate Python developer.

My main passion is web development, and I use Python for that as well. My last "cool" project was a course scheduling web application for UAEU students called Jadawil. It is still being updated every semester, but I probably won't be adding any new features. The web app is written in Flask, a web microframework for Python.

### Target Audience

This workshop assumes that you have a bit of experience working with another programming language, such as C or Java. If you don't, you may have some difficulty keeping up.

# Section 0. Setting Up the Environment

## 1. Installing Anaconda

Anaconda is a custom installer for Python that includes the most used Python libraries. It is available for all major operating systems and works pretty much the same across them all, making troubleshooting less of a problem. Also, it includes `IPython`, outlined below in step 4.

[1] Python (programming language) - http://en.wikipedia.org/wiki/Python_%28programming_language%29

To download Anaconda, visit its Downloads page. Scroll down a bit to see links to the installers. The installation process is quite straightforward. Do not change any of the options during the installation, except perhaps the installation directory.

To verify that Python was installed correctly, type `python --version` in the command prompt on Windows or the terminal on OS X/Linux. If you don't get an error, you're good to go.

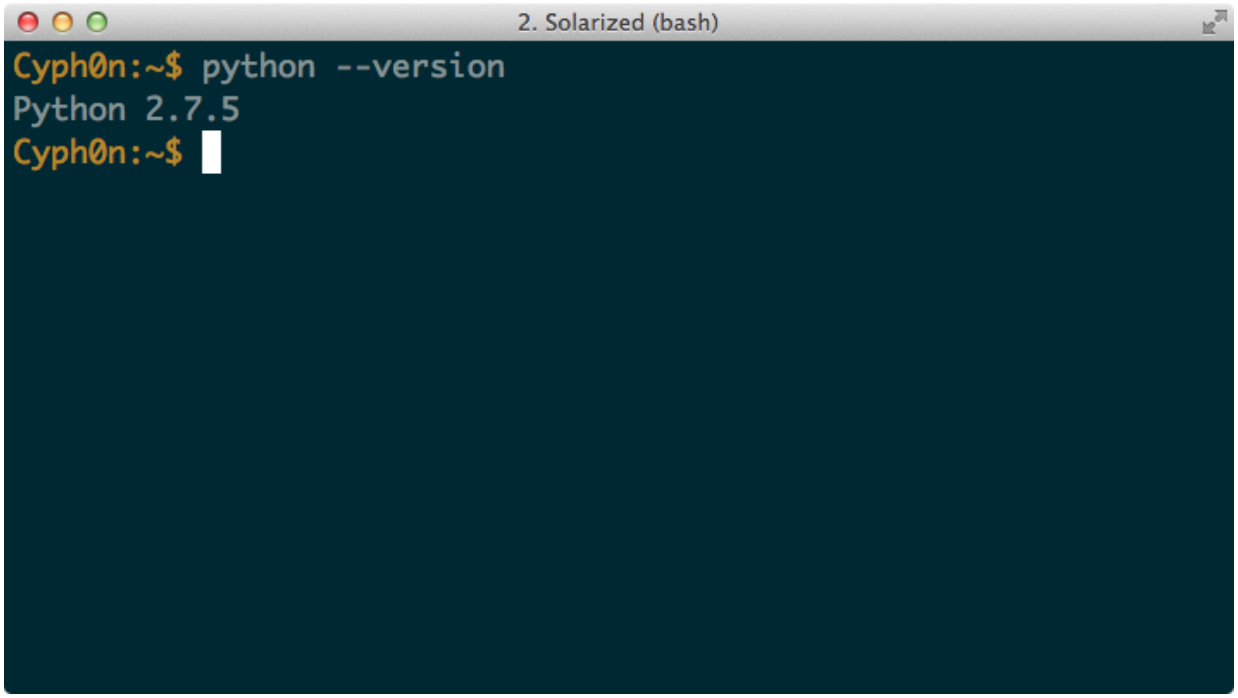


Figure 1: Python is installed correctly.

## 2. Creating a Directory

Create a directory for the workshop. We'll be saving our work in this directory. An example could be `py-workshop` on the Desktop. Navigate to this directory using your prompt's `cd` command before proceeding.

## 3. The `IPython` Notebook

`IPython` is a special version of Python that adds a good amount of useful features to the Python interpreter. In addition, it comes with a Notebook version that allows you to interactively run your code in a web browser. Since Python is a dynamic language, you do not need to compile your code - simply type the code in a block and hit `Shift-Enter` to view the results of the execution instantly.

`IPython` also allows you to include images, text, LaTeX-formatted equations, and even video along with your code *in the same notebook*. More details can be found in the `IPython` documentation.

To start up the `IPython` Notebook server, type `ipython notebook` in your terminal. This will automatically open the `IPython` Notebook dashboard page in your default web browser. To stop the server, hit `Ctrl-C`.

Create a new notebook. It will be saved in the current working directory.

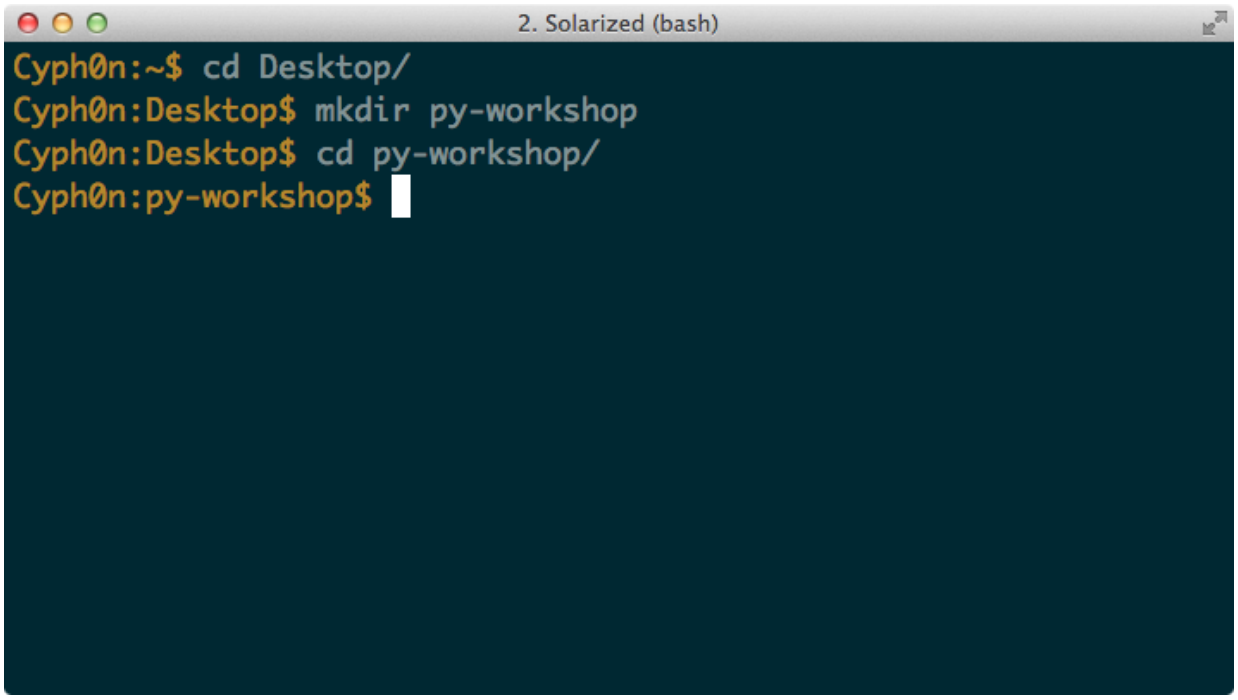


Figure 2: Creating a directory in OS X.

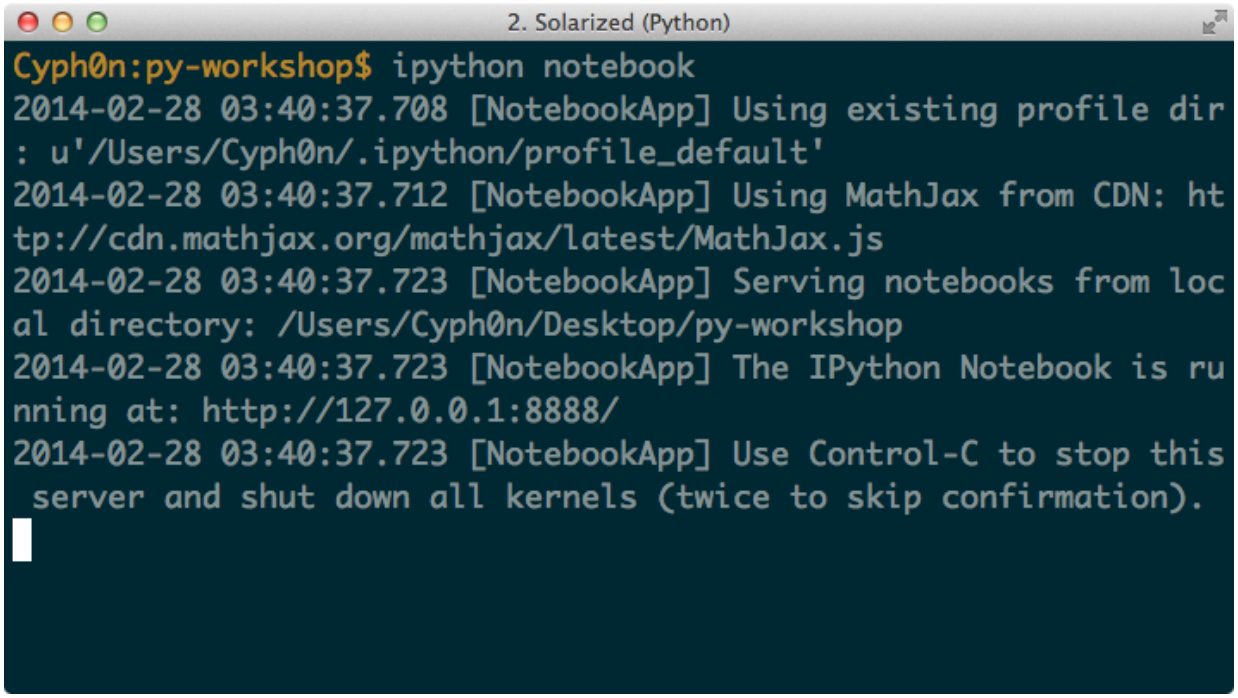


Figure 3: Running the `IPython` Notebook server.

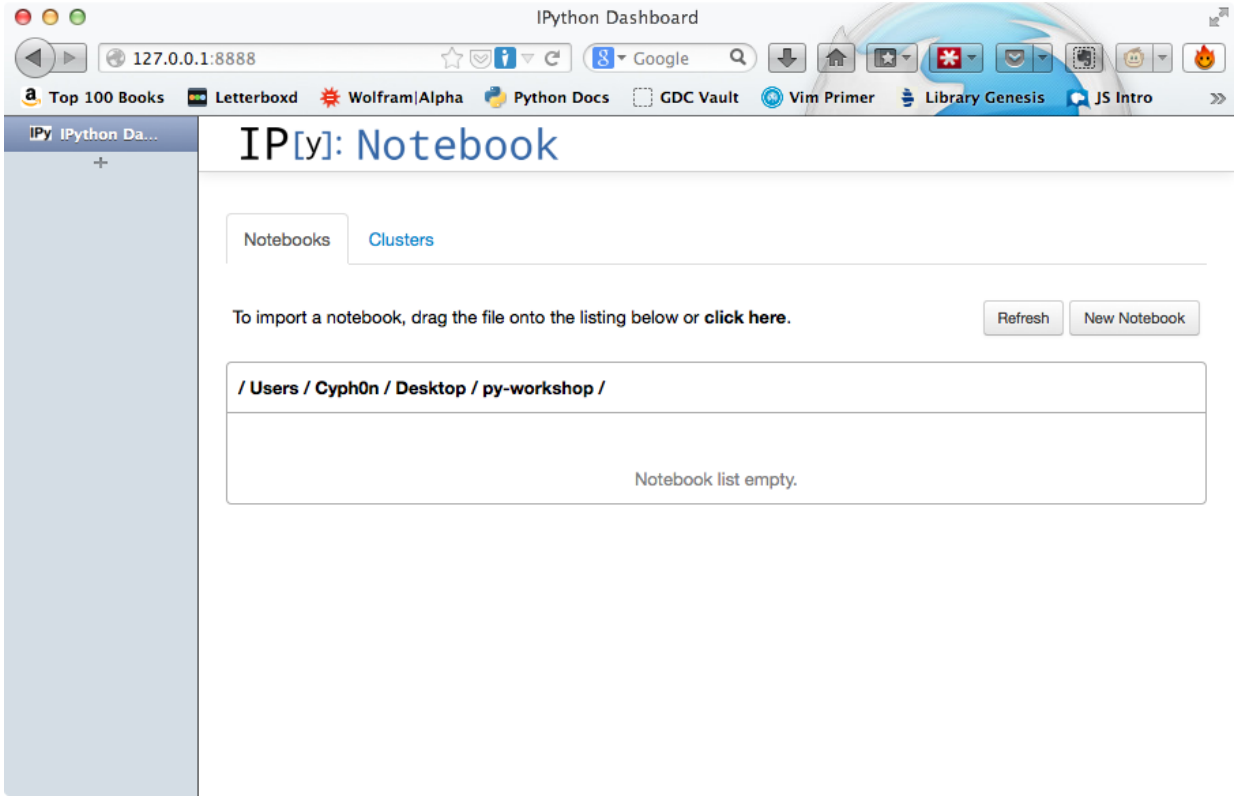


Figure 4: The `IPython` Notebook dashboard.

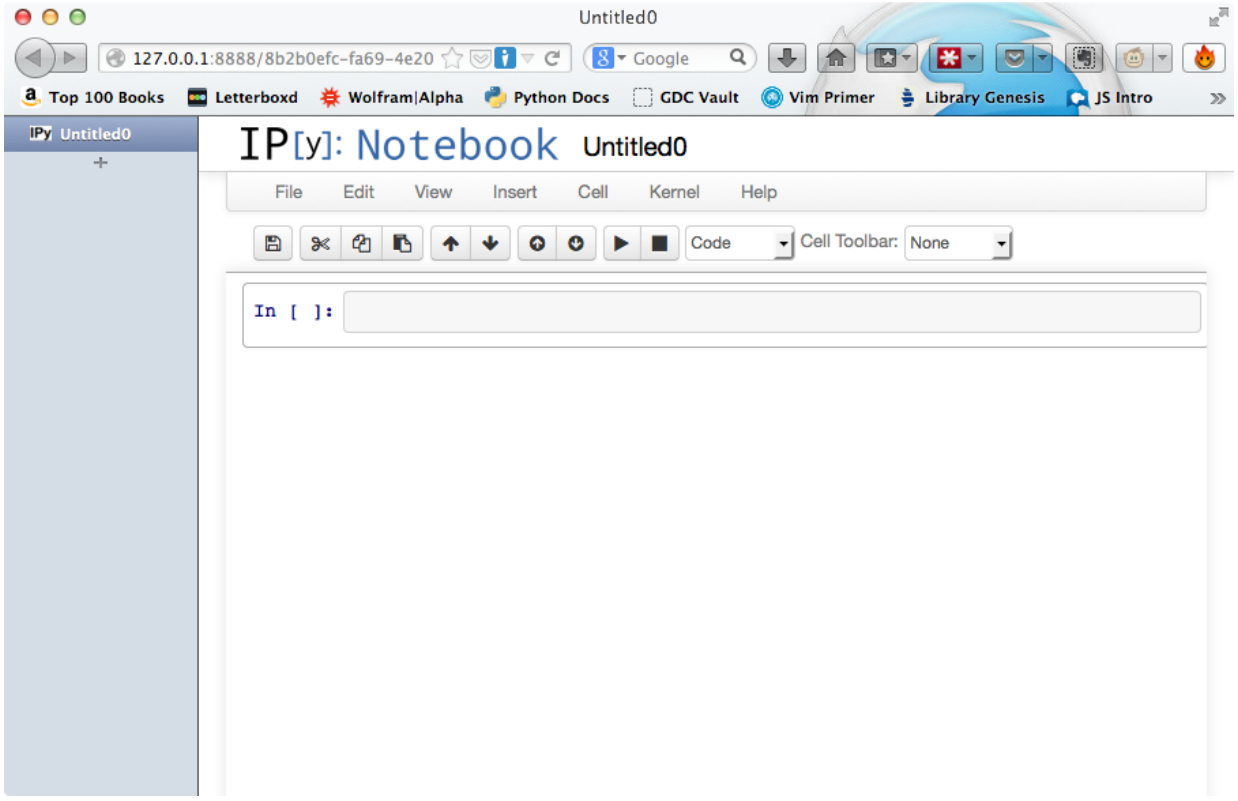


Figure 5: A new notebook.

## Section 1. Variables, Types, and User Input

Unlike in C or Java, variables in Python are not declared beforehand. This is due to dynamic typing, or "duck" typing. That means that the interpreter determines the type at runtime. However, there are types, and the main ones are `int`, `float`, `string`, `list`, and `dict`. Since Python is an object-oriented language, the types are all objects, similar to Java.

The following is some simple code that demonstrates the manipulation of variables of different types. Try to predict the output of each `print` statement before running anything. Note that string formatting in Python uses C's format specifier system.

```
# Setup some variables of different types
a = 5
b = 12.0
c = 'apple'
d = True
e = [1, 5.0, False, 'orange']

# Simple operations and access
print (a + 10) ** 2
print 'Value = %f' % (b * a)
print c + ' ' + c
print d
print e[1], e[-1]
```

In Python, taking user input is extremely simple thanks to the `raw_input` function. The function takes an optional message to display to the user and returns the entered value as a `string`. To get a number, you'll need to convert the input from `string` to `int` or `float`.

```
# Take user input
name = raw_input('Enter your name: ')
print 'Hello, %s!' % name

# Type conversion in action
num = int(raw_input())
print 'You entered %d.' % num
```

# Section 2. Flow Control and Looping

We have mentioned already that Python is a whitespace-based language, so proper indentation is extremely important. As you can see in the following examples, whitespace defines which code lies in which block. Think of it as a replacement for the curly braces in C.

## The `if` Statement

The syntax for the `if` statement is very similar to its syntax in C. The only difference is the use of `elif` instead of `else if`.

```
# Simple if-elif-else block
if a < 5:
    print 'Less.'
    print 'Still here.'
elif a == 5:
    print 'Equal.'

    if a != 5:
        print 'Impossible, right?'
else:
    print 'Greater.'
```

## The `for` Loop

The `for` loop in Python is quite different however. It is much more concise, and is used to iterate over a list instead of incrementing a counter.

The list used by the loop can be user-defined or generated using a built-in function like `range`, which generates a list of `int`s. On each iteration of the loop, the variable is assigned to an item in the list in sequential order.

```
# Append some items to the list, then iterate over it
e.append('mango')
e.append(33.5)

for item in e:
    print e

# Iterate over a range of numbers (1-10)
m = 10

for i in range(1, m+1):
    print i
```

## The `while` Loop

The `while` loop is basically the same, and is used mainly for sentinel loops in Python, as shown below.

```
# Simple while loop
i = 0
condition = True

while condition:
    if i == 5:
        condition = False
        continue

    print 'Iteration: %d' % i

    i += 1
```

# Section 3. Functions and File I/O

## Functions

Due to the dynamic typing system, functions do not need a return type or types for their argument(s). This makes writing functions much easier, since you do not have to worry about types. This can be bad in some cases, but we'll leave that for another time.

```
# Print something
def printer():
    print 'Something?'

# Add two numbers
def add(x, y):
    return x + y

# Greeting with default name
def greeting(name='world'):
    return 'Hello, %s!' % name
```

The function calling syntax is exactly the same as it is in C.

```
printer()
s = add(10, 20)
t = greeting()
u = greeting('Assil')

print s
print t
print u
```

## File Input and Output

Working with files is very easy in Python (see the pattern?), as demonstrated by the following examples. To create a new `file` object, we use the built-in `open` function. After you're done working with the file, it is advisable to call its `close` method.

```
# Create a new file in current directory and write 1-10 on seperate lines
f = open('nums.txt', 'w')

for i in range(1, 11):
    f.write('%d\n' % i)

f.close()


# Open above file for reading
f = open('nums.txt')

# Two ways to get contents of file

## Iterate over its lines
for line in f:
    print line

## Get the list of lines (includes `\n`)
lines = f.readlines()

f.close()
```

## Section 4. Imports and The Standard Library

For this part, you'll need to create a Python script. On Unix, simply type `touch test.py` in the terminal.

On Windows, you'll need to create a new file using Windows Explorer. Open the `py-workshop` folder in Explorer, right-click, and navigate to `New > Text Document`. Rename the new document to `test.py`. Make sure the extension is not `.txt`.

Type the following into `test.py`:

```
def add(x, y):
    return x + y

a = 15
b = 'apple'
```

Save `test.py` and close it. Go back to your `IPython` notebook.

We'll be using the variables and function defined in `test.py` in our notebook. This is accomplished by using the `import` statement. As you can see, the same statement is used to import built-in Python libraries, so be careful with the filename i.e. don't call your external script `math.py`.

```
# Import from the standard library
import math
from math import pi

# Import external Python script
import test

# Get variables from external script
```

```
a = test.a
b = test.b

print a, b

n = test.add(a, b)
p = math.sqrt(n)
q = math.pow(pi, 2)

print n, p, q
```

## Section 5. Project

The project combines everything covered above to create a relatively useful Python application. This application will do the following:

1. Ask the user to enter the URL of a valid website.
2. Ask the user for a path to the folder to save the above website's HTML in as a `.html` file.
3. Download the contents of the website's homepage in HTML and save it to the given file.
4. Tell the user that the process is complete and show the path to the downloaded file.

### Implementation

Before writing any code, let's write comments to define the layout of our script. It's good practice usually, and helps you organize your thinking as you work.

```
# get_html(url) -> given a URL, returns HTML content of page as a string


# Prompts for the user - URL and path


# Add filename to end of path


# Create a new HTML file at given path


# Get the webpage's contents using `get_html`


# Write HTML and then close file object


# Tell the user it has been saved and show the file's path
```

Fill in the code for each comment, starting with the prompts. Keep the `get_html` function definition until the end.

For the prompts, we need the user to enter two things: the URL and the path. That means two variables and two `raw_input` calls. Since this is a simple application, we'll leave the error handling to the user by including the input specification in each prompt.

```
# Prompts for the user
url = raw_input('Please enter a URL (without http://): ')
path = raw_input('Enter a path to save file (no trailing slash - empty for current directory): ')
```

Next, we add the filename to the end of the given path. This is done so the `open` function know's the filename as well. We'll call the file `page.html`.

```
# Add filename to end of path
full_path = path + 'page.html'
```

Now we create the HTML file using the path. Make sure to set the mode to `w` (write).

```
# Create a new HTML file at given path
f = open(full_path, 'w')
```

Here's the important part. For now, we'll just insert a call to `get_html`, which we'll define in a moment.

```
# Get the webpage's contents using `get_html`
html = get_html(url)
```

The last two parts are straightforward. Notice that string formatting can be done even before the `print` statement.

```
# Write HTML and then close file object
f.write(html)
f.close()

# Tell the user it has been saved and show the file's path
message = 'Done! File saved at: %s' % full_path
print message
```

Lastly, we need to implement `get_html`. To do that, we'll use a library included with Python called `urllib2`. It contains objects and functions that allow you to work with URLs. Obviously, we have to import the library first.

For our task, we'll be using a function called `urlopen`. It takes a URL as input, and returns a file-like object. Since `urlopen` needs a URL that starts with `http://`, we'll append it to the start of the URL. Finally, we'll invoke the `read` method of the object to get the page's HTML content as a `string` and return it.

```
import urllib2

def get_html(url):
    # Get the page's reponse
    response = urllib2.urlopen('http://' + url)

    # Get the body of the page (HTML)
    text = response.read()

    return text
```

The final result is shown below. The function does not have to be located at the top - this is simply a stylistic choice.

```
import urllib2

# get_html(url) -> given a URL, returns HTML content of page as a string
def get_html(url):
    # Get the page's reponse
    response = urllib2.urlopen('http://' + url)

    # Get the body of the page (HTML)
    text = response.read()

    return text

# Prompts for the user
url = raw_input('Please enter a URL (without http://): ')
path = raw_input('Enter a path to save file (no trailing slash - empty for current directory): ')

# Add filename to end of path
full_path = path + 'page.html'

# Create a new HTML file at given path
f = open(full_path, 'w')

# Get the webpage's contents using `get_html`
html = get_html(url)

# Write HTML and then close file object
f.write(html)
f.close()

# Tell the user it has been saved and show the file's path
message = 'Done! File saved at: %s' % full_path
print message
```

## Exercise 1: Create a Standalone Script

How about we save the code into its own Python script? Follow the steps mentioned in the previous section to create a new Python script and then copy and paste the code into it.

Assume we named the script `save_page.py`. To run it, in the command prompt (or terminal), type `python save_page.py`. You should be able to type the URL and path in the prompt and then see the result.

## Exercise 2: Take Multiple URLs From the User

First, you'll need to somehow display the two prompts multiple times to the user and save the URL and path each time. Second, you'll need to do the same procedure for each URL-path pair. **Hint:** you'll need to use loops.

I'll leave this for you to implement.

# Next Steps

## Beginner Resources

As you may have noticed, Python is an extremely vast language. For that reason, there is a lot to learn, and for a beginner especially, that can be overwhelming. To help you out, I've included free resources to take your Python to the next level.

- Learn Python the Hard Way - I believe this is where you want to start if you're serious. Its approach is quite tedious, but trust me, if you complete it, you'll be in good shape.
- The Python Tutorial - The official Python tutorial. It's a bit too cryptic for newcomers, but you should have a grasp of the fundamentals, so no problem.
- Codeacademy Python Track - A solid introduction to Python and some intermediate uses. Codeacademy also has tracks for other programming languages. A great website.
- The Python Standard Library - This is where you should go when you need to find a library or built-in function to help you accomplish a task. This should be in your browser's bookmarks toolbar.
- #python on Freenode - A great place to ask Python questions of all levels. You can connect via a web interface or through an IRC client.

## External Libraries

How about external libraries? There are a ton of them, of course. But before that, how do you install external libraries? For that, there is PyPI, the Python Package Index. It has a command line tool that simplifies the installation of such libraries. If you're using Anaconda, the `pip` tool is already installed. Just type `pip install <package-name>` to grab a package.

Below are the most well-known libraries from a variety of fields.

- NumPy - The fundamental package for scientific computing with Python. Many high-profile libraries depend on this.
- pandas - Provides high-performance, easy-to-use data structures and data analysis tools.
- SymPy - A library for working with symbolic mathematics.
- matplotlib - The standard 2D plotting library for Python. Supports MATLAB-like plotting syntax.
- Django - A powerful and complete MVC web framework used by many high-profile websites.
- Flask - My favorite web microframework. More lightweight than Django, but includes less features built-in.
- Twisted - An event-driven networking engine. Used for extremely high-performance web servers and applications.
- gevent - A co-routine based networking library. Provides tools to include lightweight concurrent threads (or "greenlets") in your applications.
- Celery - An asynchronous task queue based on distributed message passing.
- SQLAlchemy - A powerful database ORM (object relational mapper) for Python. Makes working with databases a breeze.
- Kivy - A cross-platform framework for creating NUIs (native user interfaces).
- wxPython - An API for the wxWidgets GUI development framework.

It would take quite a few pages to actually cover all of the great Python libraries, but I think the above are sufficient to demonstrate the true power and versatility of Python.

## Contact

If you have any further questions on anything Python-related, please don't hesitate to contact me via email or Twitter.

## List of Figures