Learn **Python** in Ten Minutes

The tutorial: The book

Stavros Korokithakis

Learn Python in Ten Minutes

The tutorial: The book

Stavros Korokithakis

This book is for sale at http://leanpub.com/learn-python

This version was published on 2014-01-25

Tweet This Book!

Please help Stavros Korokithakis by spreading the word about this book on Twitter!

The suggested hashtag for this book is #learn-python.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#learn-python

Contents

1 Learn Python in Ten Minutes

1.1 Preliminary fluff

So, you want to learn the Python programming language but can't find a concise and yet full-featured tutorial. This tutorial will attempt to teach you Python in ten minutes. It's probably not so much a tutorial as it is a cross between a tutorial and a cheatsheet, so it will just show you some basic concepts to start you off. Obviously, if you want to really learn a language you need to program in it for a while. I will assume that you are already familiar with programming and will, therefore, skip most of the non-language-specific stuff. Also, pay attention because, due to the terseness of this tutorial, some things will be introduced directly in code and only briefly commented on.

1.2 Properties

Python is strongly typed (i.e. types are enforced), dynamically, implicitly typed (i.e. you don't have to declare variables), case sensitive (i.e. var and VAR are two different variables) and object-oriented (i.e. everything is an object).

1.3 Getting help

Help in Python is always available right in the interpreter. If you want to know how an object works, all you have to do is call help(<object>)! Also useful are dir(), which shows you all the object's methods, and <object>.__doc__, which shows you its documentation string:

> help(5) Help on int object: (etc etc)
>
> dir(5) ['**abs**', '**add**', ...]
>
> abs.**doc** 'abs(number) -> number

Return the absolute value of the argument.'

1.4 Syntax

Python has no mandatory statement termination characters and blocks are specified by indentation. Indent to begin a block, dedent to end one. Statements that expect an indentation level end in a colon (:). Comments start with the pound (#) sign and are single-line, multi-line strings are used for multi-line comments. Values are assigned (in fact, objects are bound to names) with the _equals_ sign ("="), and equality testing is done using two _equals_ signs ("=="). You can increment/decrement values using the += and -= operators respectively by the right-hand amount. This works on many datatypes, strings included. You can also use multiple variables on one line. For example:

```
 myvar = 3
>>> myvar += 2
>>> myvar
5
>>> myvar -= 1
>>> myvar
4
"""This is a multiline comment.
The following lines concatenate the two strings."""
>>> mystring = "Hello"
>>> mystring += " world."
>>> print mystring
Hello world.
# This swaps the variables in one line(!).
# It doesn't violate strong typing because values aren't
# actually being assigned, but new objects are bound to
# the old names.
>>> myvar, mystring = mystring, myvar
```

1.5 Data types

The data structures available in python are lists, tuples and dictionaries. Sets are available in the `sets` library (but are built-in in Python 2.5 and later). Lists are like one-dimensional arrays (but you can also have lists of other lists), dictionaries are associative arrays (a.k.a. hash tables) and tuples are immutable one-dimensional arrays (Python "arrays" can be of any type, so you can mix e.g. integers, strings, etc in lists/dictionaries/tuples). The index of the first item in all array types is 0. Negative numbers count from the end towards the beginning, -1 is the last item. Variables can point to functions. The usage is as follows:

```
sample = [1, ["another", "list"], ("a", "tuple")]
>>> mylist = ["List item 1", 2, 3.14]
>>> mylist[0] = "List item 1 again"
>>> mylist[-1] = 3.14
>>> mydict = {"Key 1": "Value 1", 2: 3, "pi": 3.14}
>>> mydict["pi"] = 3.15
>>> mytuple = (1, 2, 3)
>>> myfunction = len
>>> print myfunction(mylist)
3
```

You can access array ranges using a colon (:). Leaving the start index empty assumes the first item, leaving the end index assumes the last item. Negative indexes count from the last item backwards (thus -1 is the last item) like so:

```
mylist = ["List item 1", 2, 3.14]
>>> print mylist[:]
['List item 1', 2, 3.1400000000000001]
>>> print mylist[0:2]
['List item 1', 2]
>>> print mylist[-3:-1]
['List item 1', 2]
>>> print mylist[1:]
[2, 3.14]
```

1.6 Strings

Its strings can use either single or double quotation marks, and you can have quotation marks of one kind inside a string that uses the other kind (i.e. “He said ‘hello’.” is valid). Multiline strings are enclosed in _triple double (or single) quotes_ (“””). Python supports Unicode out of the box, using the syntax u”This is a unicode string”. To fill a string with values, you use the % (modulo) operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions, like so:

```
print "Name: %s\
Number: %s\
String: %s" % (myclass.name, 3, 3 * "-")
Name: Poromenos
Number: 3
String: ---

strString = """This is
a multiline
string."""

# WARNING: Watch out for the trailing s in "%(key)s".
>>> print "This %(verb)s a %(noun)s." % {"noun": "test", "verb": "is"}
This is a test.
```

1.7 Flow control statements

Flow control statements are `if`, `for`, and `while`. There is no `select`; instead, use if. Use for to enumerate through members of a list. To obtain a list of numbers, use `range(<number>)`. These statements' syntax is thus:

```
rangelist = range(10)
>>> print rangelist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing
```

```
if rangelist[1] == 2:
    print "The second item (lists are 0-based) is 2"
elif rangelist[1] == 3:
    print "The second item (lists are 0-based) is 3"
else:
    print "Dunno"

while rangelist[1] == 1:
    pass
```

1.8 Functions

Functions are declared with the “def” keyword. Optional arguments are set in the function declaration after the mandatory arguments by being assigned a default value. For named arguments, the name of the argument is assigned a value. Functions can return a tuple (and using tuple unpacking you can effectively return multiple values). Lambda functions are ad hoc functions that are comprised of a single statement. Parameters are passed by reference, but immutable types (tuples, ints, strings, etc) *cannot be changed*. This is because only the memory location of the item is passed, and binding another object to a variable discards the old one, so immutable types are replaced. For example:

```
# Same as def f(x): return x + 1
functionvar = lambda x: x + 1
>>> print functionvar(1)
2

# an_int and a_string are optional, they have default values
# if one is not passed (2 and "A default string", respectively).
def passing_example(a_list, an_int=2, a_string="A default string"):
    a_list.append("A new item")
    an_int = 4
    return a_list, an_int, a_string

>>> my_list = [1, 2, 3]
>>> my_int = 10
>>> print passing_example(my_list, my_int)
([1, 2, 3, 'A new item'], 4, "A default string")
>>> my_list
[1, 2, 3, 'A new item']
>>> my_int
10
```

1.9 Classes

Python supports a limited form of multiple inheritance in classes. Private variables and methods can be declared (by convention, this is not enforced by the language) by adding at least two leading underscores and at most one trailing one (e.g. "__spam"). We can also bind arbitrary names to class instances. An example follows:

```
class MyClass(object):
    common = 10
    def __init__(self):
        self.myvariable = 3
    def myfunction(self, arg1, arg2):
        return self.myvariable

    # This is the class instantiation
>>> classinstance = MyClass()
>>> classinstance.myfunction(1, 2)
3
# This variable is shared by all classes.
>>> classinstance2 = MyClass()
>>> classinstance.common
10
>>> classinstance2.common
10
# Note how we use the class name
# instead of the instance.
>>> MyClass.common = 30
>>> classinstance.common
30
>>> classinstance2.common
30
# This will not update the variable on the class,
# instead it will bind a new object to the old
# variable name.
>>> classinstance.common = 10
>>> classinstance.common
10
>>> classinstance2.common
30
>>> MyClass.common = 50
# This has not changed, because "common" is
# now an instance variable.
```

```
>>> classinstance.common
10
>>> classinstance2.common
50

# This class inherits from MyClass. The example
# class above inherits from "object", which makes
# it what we call a "new-style class".
# Multiple inheritance is declared as:
# class OtherClass(MyClass1, MyClass2, MyClassN)
class OtherClass(MyClass):
    # The "self" argument is passed automatically
    # and refers to the class instance, so you can set
    # instance variables as above, but from inside the class.
    def __init__(self, arg1):
        self.myvariable = 3
        print arg1

>>> classinstance = OtherClass("hello")
hello
>>> classinstance.myfunction(1, 2)
3
# This class doesn't have a .test member, but
# we can add one to the instance anyway. Note
# that this will only be a member of classinstance.
>>> classinstance.test = 10
>>> classinstance.test
10
```

1.10 Exceptions

Exceptions in Python are handled with try-except [exceptionname] blocks:

```
def some_function():
    try:
        # Division by zero raises an exception
        10 / 0
    except ZeroDivisionError:
        print "Oops, invalid."
    else:
        # Exception didn't occur, we're good.
        pass
    finally:
        # This is executed after the code block is run
        # and all exceptions have been handled, even
        # if a new exception is raised while handling.
        print "We're done with that."

>>> some_function()
Oops, invalid.
We're done with that.
```

1.11 Importing

External libraries are used with the `import [libname]` keyword. You can also use `from [libname] import [funcname]` for individual functions. Here is an example:

```
import random
from time import clock

randomint = random.randint(1, 100)
>>> print randomint
64
```

1.12 File I/O

Python has a wide array of libraries built in. As an example, here is how serializing (converting data structures to strings using the `pickle` library) with file I/O is used:

```
import pickle
mylist = ["This", "is", 4, 13327]
# Open the file C:\\binary.dat for writing. The letter r before the
# filename string is used to prevent backslash escaping.
myfile = open(r"C:\\binary.dat", "w")
pickle.dump(mylist, myfile)
myfile.close()

myfile = open(r"C:\\text.txt", "w")
myfile.write("This is a sample string")
myfile.close()

myfile = open(r"C:\\text.txt")
>>> print myfile.read()
'This is a sample string'
myfile.close()

# Open the file for reading.
myfile = open(r"C:\\binary.dat")
loadedlist = pickle.load(myfile)
myfile.close()
>>> print loadedlist
['This', 'is', 4, 13327]
```

1.13 Miscellaneous

- Conditions can be chained. `1 < a < 3` checks that a is both less than 3 and greater than 1.
- You can use `del` to delete variables or items in arrays.
- List comprehensions provide a powerful way to create and manipulate lists. They consist of an expression followed by a `for` clause followed by zero or more `if` or `for` clauses.

An example:

```
lst1 = [1, 2, 3]
>>> lst2 = [3, 4, 5]
>>> print [x * y for x in lst1 for y in lst2]
[3, 4, 5, 6, 8, 10, 9, 12, 15]
>>> print [x for x in lst1 if 4 > x > 1]
[2, 3]
# Check if an item has a specific property.
# "any" returns true if any item in the list is true.
>>> any([i % 3 for i in [3, 3, 4, 4, 3]])
True
# This is because 4 % 3 = 1, and 1 is true, so any()
# returns True.

# Check how many items have this property.
>>> sum(1 for i in [3, 3, 4, 4, 3] if i == 4)
2
>>> del lst1[0]
>>> print lst1
[2, 3]
>>> del lst1
```

Global variables are declared outside of functions and can be read without any special declarations, but if you want to write to them you must declare them at the beginning of the function with the “global” keyword, otherwise Python will bind that object to a new local variable (be careful of that, it’s a small catch that can get you if you don’t know it). For example:

```
number = 5

def myfunc():
    # This will print 5.
    print number

def anotherfunc():
    # This raises an exception because the variable has not
    # been bound before printing. Python knows that it an
    # object will be bound to it later and creates a new, local
    # object instead of accessing the global one.
    print number
    number = 3

def yetanotherfunc():
    global number
    # This will correctly change the global.
    number = 3
```

1.14 Epilogue

This tutorial is not meant to be an exhaustive list of all (or even a subset) of Python. Python has a vast array of libraries and much much more functionality which you will have to discover through other means, such as the excellent book Dive into Python. I hope I have made your transition in Python easier. Please let me know if you believe there is something that could be improved or added or if there is anything else you would like to see (classes, error handling, anything).

2 Learn Python in More Minutes

2.1 Extra fluff

After the amazing success of the tutorial, with more than *one and a half million* readers since it was first written (give or take a million, anyway), I decided to publish it as a book. Since the book was well-received, I decided to add some extra content to help the lucky few, who have more than ten minutes, learn more Python.

I'm not very good with rambling, or even with being verbose, so the extended section will still be as short as necessary to explain new concepts adequately.

The following sections will probably also be made available as posts on my blog[1], for your reading pleasure.

2.2 Developing and deploying Python programs

So, you've written an amazing Python program, it does everything you want and then some. You stop and stare at it in amazement, and realize that it's just *so* great, that it would be selfish not to share it with the entire world.

However, there's one small problem. Your program uses various libraries, which your users will need to install, and you want to make that process as painless as possible, or, if it's a web app, you want to put it up on a server for everyone to enjoy, but you need to figure out what libraries it uses and install those on the server as well, which is a hassle. What's a good way to ease this pain?

Suppose that your first program has made you a millionaire, and now you're running low on cash, so you decide to spend a few hours writing another one. However, your system still has all the old versions of the libraries you used to build the first program, and when you try to upgrade them, it breaks because it hasn't been maintained in ages. How do you solve that problem?

Luckily, there's an easy way to solve both problems. We can use `virtualenv` to create separate, self-contained virtual environments where each environment is contained in a its own directory. Each environment can be activated when you want to work on the program that uses it.

The best way to show how `virtualenv` works is by example, so let's go through the process of installing and using it. Assuming you have the average Python installation (probably 2.7, on any of the three main OSes), the following steps should work as they are. Windows might do some things slightly differently, but mostly it should all work the same. In the examples, the lines starting with

[1] http://www.korokithakis.net

an "$" is what you will need to type in ($ is my prompt), so ignore the $ and type in the rest of the command.

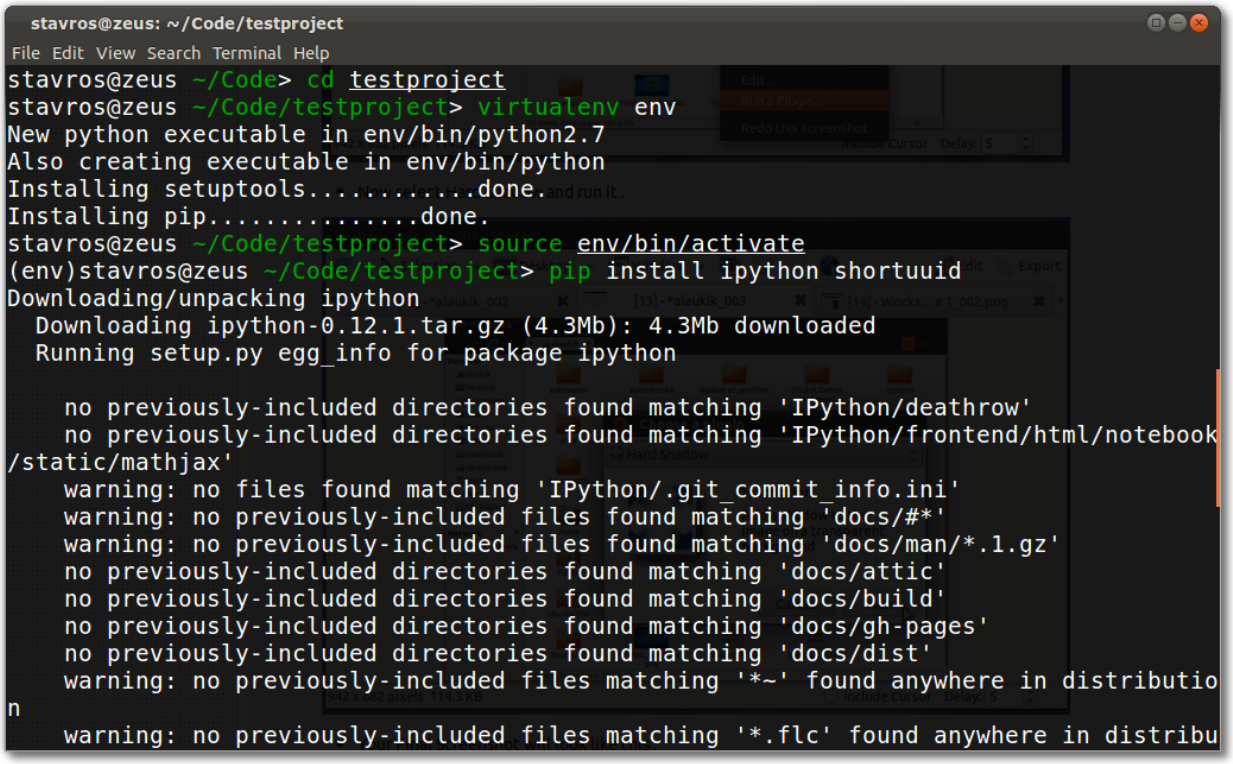
To install `virtualenv` (and `pip`, while we're at it), you can use setuptools:

```
$ easy_install pip virtualenv
```

After that, both packages will be installed globally in your system, so you might want to use `sudo` if you are on Linux or OS X. We can now continue to creating an environment. I prefer to put the environment in the directory that contains the project I'm working on, so switch to the directory that contains your scripts and run (you shouldn't need to use `sudo`):

```
$ virtualenv env
```

That will create a directory called `env` in your current directory and put the environment there. If you look inside, you should see some subdirectories, containing various files. The `bin` directory, in particular, should contain executables called `python`, `pip`, `easy_install`, etc. These executables differ from their globally-installed namesakes in that the former will be run in the virtual environment you've just created, while the latter will run system-wide.


stavros@zeus: ~/Code/testproject
File Edit View Search Terminal Help

```
stavros@zeus ~/Code> cd testproject
stavros@zeus ~/Code/testproject> virtualenv env
New python executable in env/bin/python2.7
Also creating executable in env/bin/python
Installing setuptools............done.
Installing pip...............done.
stavros@zeus ~/Code/testproject> source env/bin/activate
(env)stavros@zeus ~/Code/testproject> pip install ipython shortuuid
Downloading/unpacking ipython
  Downloading ipython-0.12.1.tar.gz (4.3Mb): 4.3Mb downloaded
  Running setup.py egg_info for package ipython

    no previously-included directories found matching 'IPython/deathrow'
    no previously-included directories found matching 'IPython/frontend/html/notebook
/static/mathjax'
    warning: no files found matching 'IPython/.git_commit_info.ini'
    warning: no previously-included files found matching 'docs/#*'
    warning: no previously-included files found matching 'docs/man/*.1.gz'
    no previously-included directories found matching 'docs/attic'
    no previously-included directories found matching 'docs/build'
    no previously-included directories found matching 'docs/gh-pages'
    no previously-included directories found matching 'docs/dist'
    warning: no previously-included files matching '*~' found anywhere in distributio
n
    warning: no previously-included files matching '*.flc' found anywhere in distribu
```

Creating a virtualenv and installing packages in it

To illustrate this point, we need to install some packages. To do this, we will use `pip`, because it's great. We already installed it earlier, so we can use it simply by doing:

```
$ env/bin/pip install ipython shortuuid
```

`ipython` is a very nice Python shell, which you might already be familiar with, and `shortuuid` is a small library for creating short unique IDs, which we'll use to demonstrate how virtualenv works.

You might have noticed that we didn't just run `pip`, we ran `env/bin/pip`, which means that the two packages are now installed inside the virtualenv. Sure enough, if you run:

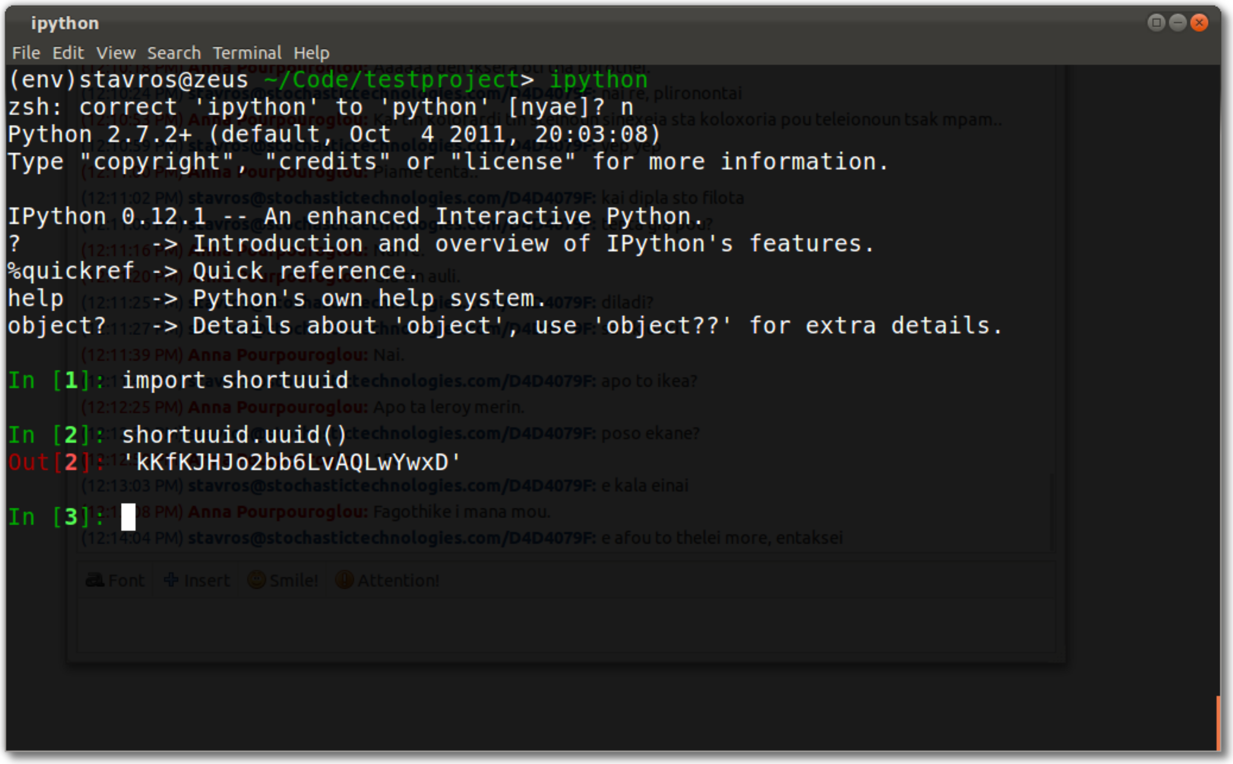
```
$ env/bin/ipython
```

you will see that the ipython shell opens up. If you try to `import shortuuid`, you will see the following:

```
IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import shortuuid

In [2]: shortuuid.uuid()
Out[2]: 'ogxPuKH7qvtoXSzNmVECW'
```



```
(env)stavros@zeus ~/Code/testproject> ipython
zsh: correct 'ipython' to 'python' [nyae]? n
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import shortuuid

In [2]: shortuuid.uuid()
Out[2]: 'kKfKJHJo2bb6LvAQLwYwxD'

In [3]:
```

shortuuid is now available

You will get a different ID, but the gist is the same. `shortuuid` has been installed and is working properly. If you now type `exit` to exit ipython, run the system-wide python installation (by typing `python`), and try to import `shortuuid`, what do you think will happen? Let's try it out:

```
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shortuuid
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named shortuuid
>>>
```

This is because, as I mentioned before, these packages have only been installed in the virtualenv directory, and we didn't run the Python interpreter that's inside the virtualenv, we ran the system-wide one. These packages don't exist globally in the system, so we can be sure that whatever package we install will be in its own, tidy little directory.

What happens if there's already a package installed system-wide and we try to install it in the virtualenv, or even if we *don't* install it in the virtualenv? Can we still access it? The answer is "no, we cannot", because virtualenv, by default, restricts all access to system packages. This can be changed when creating the virtualenv, but it's beyond our scope now. As far as the virtualenv is concerned, the system has no packages other than the ones actually installed by us in the virtualenv. If a library is at version X globally, but we install version Y in the virtualenv, programs in that virtualenv will only be able to see version Y.

So, we have installed our packages and are able to write our program knowing that we won't be messing our computer up with the various packages, but using `env/bin/python` all the time is a bit of a hassle. To activate one virtualenv to work with *for one terminal session*, we can run:

```
$ source env/bin/activate
```

This should put the environment's name somewhere on our prompt, and, from now on, any program we run that's installed in the virtualenv, will be run from it. For example, if we run `python`, or `pip`, or `ipython`, we will actually be running the versions inside the environment, so they will be able to see (and install or remove, in the case of pip) only the packages inside the environment.

You might be thinking "This is all very nice so far, but you said I could easily distribute my program, and I see none of that!", and you would be right. Let's see how we can obtain a list of all the packages that are installed in the current virtualenv. Fortunately, `pip` makes this trivial:

```
$ pip freeze
ipython==0.12.1
shortuuid==0.2
```

You can see that `pip` gave us a list of all the packages we have installed, isn't that fantastic? Yes, of course it is. We can get that list and stick it in a text file (I like to call it requirements.txt). We can then send that file to a friend, along with the source code, and all they have to do to ensure that our program will run is create a virtualenv and install the packages we've provided, with three simple commands:

```
$ virtualenv env
$ source env/bin/activate
$ pip install -r requirements.txt
```

This will create the virtualenv and install all the packages we have specified in the requirements file, at the specific versions we need, no less. You're happy, your friend is happy, I'm happy, everyone's happy.

That's not all `pip` can do, though. It can remove packages, install them directly from source directories, from remote repositories, just download (but not install) them, and much more, all for one low, low price. To get a better view of what `pip` (or `virtualenv`) can do, just look at their respective help sections:

```
$ pip --help
$ virtualenv --help
```

They contain much more useful functionality, so feel free to explore around. You can get rid of virtualenvs just by deleting the directory, absolutely nothing else in your system is touched in any way.