```
#!/usr/bin/env python
"""
The first line in this file is the "shebang" line.  When you execute a file
from the shell, the shell tries to run the file using the command specified
on the shebang line.  The ! is called the "bang".  The # is not called the
"she", so sometimes the "shebang" line is also called the "hashbang".
The hash character is used because it defines a comment in most scripting
languages, so the shebang line will be ignored by the scripting language
by default.
The shebang line was invented because scripts are not compiled, so they are
not executable files, but people still want to "run" them.  The shebang
line specifies exactly how to run a script.  In other words, this shebang
line says that, when I type in ./basics.py, the shell will actuall run
  /usr/bin/env python basics.py
We use
  #!/usr/bin/env python
instead of
  #!/usr/bin/python
because we must specify an absolute path to a program, and /usr/bin/env
is a utility that uses the user's path to run an application (in this
case, python).  Thus, it's more portable.

More on shebang lines, including portability:
http://en.wikipedia.org/wiki/Shebang_(Unix)

If you don't like this basic walk through python, check out
http://docs.python.org/tutorial/
or
http://diveintopython.org/

In order to execute a python script without explicitly running python,
you need to add execute permissions to the file.  To do add execute permission
to basics.py, use
  chmod u+x basics.py

Beware that, while I have programmed in python, I am by no means an expert.
If anything in this file has bad style or is wrong, let me know!
Sam King <samking@cs.stanford.edu>
Made in Spring 2011 for CS1U at Stanford.
http://creativecommons.org/licenses/by/3.0/

This is a multiline string, which can also act as a mutliline comment.
"""

# This is a single line comment

#sys lets you access parameters
import sys
#os lets you interact with the os, including running shell scripts
import os



""" This is a multiline string, which can also act
as a multiline comment """

print """ any line of code that I put out here will be executed when the
script is run.  If you want to start at the main function like in c++ or java,
look at the boilerplate code at the bottom.  """

def basics():
  """A mulitline comment that is defined right below a function will be
  the comment for that function.  This lets automatic tools know that it's the
  function's comment."""
  print """
def basics(): defines the function \"basics\".
Note the colon.  A colon is used before any time when you would indent (loop,
if, function).
Whitespace is significant, so you must indent.  Also, mixing tabs and spaces
```

```
is a bad idea.
If you want to paste your code into a python interpreter to test, the
interpreter will think that you are done with a code block if you ever have a
blank line (even though it will be fine when you're writing a script), so keep
that in mind before adding extra whitespace

for loops use colons and have whitespace also.  If you're used to programming
in c or java, you might not be used to the foreach loop.  In python, every for
loop is a foreach loop.  That means that you say
  for element in collection:
and "element" will assume every value in the collection.  So, if you have an
array with the elements in the range between 0 and 9, then you will execute
the loop once with element as every value between 0 and 9, inclusive. Thus,
  for i in range(10):
    print i
will emulate the following c++ code:
  for (int i = 0; i < 10; i++) {
    printf("%d\\n", i);
  }
while loops are as expected

"""
  print "this is a for loop!"
  for i in range(10):
    print i
  print "now as a while loop!"
  i = 0
  while (i < 10):
    print str(i) + ' ' , #a comma here means that print won't add a newline
    i += 1 #note: there's no i++ in python.  You have to use += 1.  Sorry.
  print
  print """
variables are dynamically typed.  So,
  foo = 0
will assign foo to be 0.
  foo = "hello, world"
will assign foo to be the string "hello, world".
  foo = [1, 2]
will assign foo to be an array with the values 1 and 2
  foo.append(1)
will change foo to [1, 2, 1]
  foo[0] = 9
will change foo to [9, 2, 1]
  foo = {'hello':'world', 1:'dynamic', 2:3}
will assign foo to be a dictionary (a map) from 'hello' to 'world', the
integer 1 to the string 'dynamic', and the integer 2 to the integer 3.
  foo['foo'] = 'bar'
will then assign a new mapping from 'foo' to 'bar' in foo.
  foo = set()
will make foo a set
  foo.add('bar')
will make foo a set containing 'bar'

All data in python are objects.  Python also knows how to print out its
built in objects out of the box.  So, if you try to print an array or
you're at a python interactive shell and type out the array, it will print.
Python passes referneces to objects by value (the same as java).
This can seem confusing or unintuitive because some objects are mutable
whereas others are immutable.  Strings, numbers, and tuples (a tuple is
like a low-powered array) are immutable, which means that they're as good
as passed by value.
"""


def file_reading(filename):
  """Demonstrates how to read a file the provided name"""
  #open a file for reading
  file = open(filename, 'r')
  #this turns the file into an array.
  lines = file.readlines()
  for line in lines:
    print line
  #or, you could just print the file directly without reading it into an array
  for line in file:
    print line
  #or, you could read the whole file into a string
  file_str = file.read()
  #and then split up the string based on whitespace
  words = file_str.split()
  file.close()
  #open a file for writing
  outfile = open(filename + ".output", 'w')
  outfile.write("This file has text in it now!")
  outfile.close()


def logic():
  """Demonstrates if, elif, else, and, or, ==, !=, <, >, etc"""
  if 1 == 1 and 1 == 2 and 1 > 2 and 2 < 1:
    print "Math is wrong."
  #since indentation is important in python, it uses elif instead of else if
  elif 1 >= 2 or 1 != 2:
    print "Math is okay."
  else:
    print "Math is still wrong."

#a basic data structure class
class SomeDataStructure:
  def __init__(self, x, y, foo="bar"):
    """when DataStructure is constructed, init will run.  Since it is a data
    structure, it stores the provided x, y, and foo as instance variables x, y,
    and foo.  self is kind of like a "this" pointer.  In this example, foo has a
    default value, "bar".  You can use default values in any method. """
    self.x = x
    self.y = y
    self.foo = foo

  def __repr__(self):
    """repr mens representation. When you print an object, this method is called.
    The java equivalent is toString"""
    return repr(self.x) + ", " + repr(self.y) + ", " + repr(self.foo)

  def set_y(self, y):
    self.y = y

def classes():
  #instantiate a user made class
  data = SomeDataStructure(7, 9, "hey")
  #the third argument is optional
  more_data = SomeDataStructure(1, 2)
  #we defined a repr, so we can print out our data structures without any work
  print data
  print more_data
  #access instance variables like this
  more_data.x = 3
  print more_data
  #access instance methods like this
  more_data.set_y(9)
  print more_data

def method_with_defaults(foo="foo", bar="baz", jump=True, how_high=10, null=None):
  """All of the arguments have defaults, so you can call it with no arguments,
by naming arguments, or by providing some arguments in order"""
  print "Foo: " + foo + " Bar: " + bar + " Jump: " + str(jump) + \
      " How High: " + str(how_high) + " Null: " + str(null)

def default_values():
  #you can provide no arguments, and everything will be default
  method_with_defaults()
  #you can provide values for the first few arguments, and the rest will be default
  method_with_defaults("arg1", "arg2")
  #you can say name=value, and then the named argument will get the provided value
  method_with_defaults(null="Null should by called \"null\" in Python")

def main():
  if len(sys.argv) != 2 or sys.argv[1] != "-h":
    print "usage: python basics.py -h"
    sys.exit(1)

  basics()
  file_reading('foo')
  logic()
  classes()
  default_values()
  print """
PS - in case you actually ran this program, you should look at the
source.  The source has some comments and code that you should read."""

# Standard boilerplate to call main().
if __name__ == '__main__':
  main()
```