**See It to Believe It**

Until now, the Python code you've been writing comes from one source and only goes to one place: you type it in at the keyboard and its results are displayed in the console. But what if you want to read information from a file on your computer, and/or write that information to another file?

This process is called *file I/O* (the "I/O" stands for "input/output"), and Python has a number of built-in functions that handle this for you.

Check out the code in the editor to the right.

**Instructions**

**1.**

Click Run! You just wrote all the contents of my\_list to a text file called output.txt.

|  |
| --- |
| my\_list = [i \*\* 2 for i in range(1, 11)]  # Generates a list of squares of the numbers 1 - 10  f = open("output.txt", "w")  for item in my\_list:  f.write(str(item) + "\n")  f.close() |

|  |
| --- |
| 1  4  9  16  25  36  49  64  81  100 |

# The open() Function

Let's walk through the process of writing to a file one step at a time.

The first code that you saw executed in the previous exercise was this:

f = open("output.txt", "w")

This told Python to open output.txtin "w" mode ("w" stands for "write"). We stored the result of this operation in a file object, f.

Doing this opens the file in write-mode and prepares Python to send data into the file.

**Instructions**

**1.**

Create a variable, my\_file, and set it equal to calling the open() function on output.txt. In this case, pass "r+" as a second argument to the function so the file will allow you to read and write to it! (See the Hint for details.)

Hint

You can open files in any of the following modes:

* write-only mode ("w")
* read-only mode ("r")
* read and write mode ("r+")
* append mode ("a"), which adds any new data you write to the file to the end of the file.

|  |
| --- |
| my\_file = open("output.txt", "r+") |

# Writing

Good work! Now it's time to write some data to a new .txt file.

We added the list comprehension from the first exercise to the code in the editor. Our goal in this exercise will be to write each element of that list to a file called output.txt. The output.txt file that you write to will be created in your current folder - for simplicity, the folder has been hidden. output.txt will list each number on its own line.

We can write to a Python file like so:

my\_file.write("Data to be written")

The .write() method takes a string argument, so we'll need to do a few things here:

You must close the file. You do this simply by calling my\_file.close() (we did this for you in the last exercise). If you don't close your file, Python won't write to it properly. From here on out, you gotta close your files!

**Instructions**

**1.**

Iterate over my\_list to get each value.

Use my\_file.write() to write each value to a text file called, output.txt.

Make sure to call str() on the iterating data so .write() will accept it.

Make sure to add a newline (+ "\n") after each element to ensure each will appear on its own line.

Use my\_file.close() to close the file when you're done.

Passing the exercise means that you successfully wrote my\_list to output.txt!

Hint

Remember, the syntax for iterating over a list looks like this:

for item in list: # Do something

You should write to the file inside your iterator, but you should close the file outsideyour iterator—otherwise, you'll attempt to close the file after you write each line!

And the syntax for calling str() looks like this:

str(42) # => "42"

|  |
| --- |
| my\_list = [i \*\* 2 for i in range(1, 11)]  my\_file = open("output.txt", "w")  # Add your code below!  for item in my\_list:  my\_file.write(str(item)+"\n")  my\_file.close() |

# Reading

Excellent! You're a pro.

Finally, we want to know how to read from our output.txt file. As you might expect, we do this with the read() function, like so:

print my\_file.read()

**Instructions**

**1.**

Declare a variable, my\_file, and set it equal to the file object returned by calling open() with both "output.txt" and "r".

Next, print the result of using .read() on my\_file, like the example above.

Make sure to .close() your file when you're done with it! All kinds of doom will happen if you don't.

Hint

Remember, the syntax for opening a file looks like this:

variable = open("filename", "mode")

|  |
| --- |
| my\_file = open("output.txt", "r")  print my\_file.read()  my\_file.close() |
| 1  4  9  16  25  36  49  64  81  100 |

# Reading Between the Lines

What if we want to read from a file line by line, rather than pulling the entire file in at once. Thankfully, Python includes a .readline()method that does exactly that.

If you open a file and call .readline()on the file object, you'll get the first line of the file; subsequent calls to .readline() will return successive lines.

**1.**

Declare a new variable my\_file and store the result of calling open() on the "text.txt" file in "r"ead-only mode.

On three separate lines, printout the result of calling my\_file.readline(). See how it gets the next line each time?

Don't forget to .close() your file when you're done with it!

Hint

Remember, to open a file:

variable = open("filename", "mode")

To read from the file, you can just call variable.readline(). Make sure to close your file once you're done reading from it!

|  |
| --- |
| my\_file = open("text.txt","r")  print my\_file.readline()  print my\_file.readline()  print my\_file.readline()  my\_file.close() |
| I'm the first line of the file!  I'm the second line.  Third line here, boss. |

**PSA: Buffering Data**

We keep telling you that you always need to close your files after you're done writing to them. Here's why!

During the I/O process, data is *buffered*: this means that it is held in a temporary location before being written to the file.

Python doesn't *flush the buffer*—that is, write data to the file—until it's sure you're done writing. One way to do this is to close the file. If you write to a file without closing, the data won't make it to the target file.

**Instructions**

**1.**

Check out our *extremely bad*code in the editor. Click Run—you'll note that our read\_file.read() didn't read any data back!

* Add a write\_file.close()call after writing to the file but before reading it.
* Add a read\_file.close()call after the print read\_file.read() line
* Run the code again.
* This time, you'll see the data come through!

|  |
| --- |
| * # Use a file handler to open a file for writing * write\_file = open("text.txt", "w") * # Open the file for reading * read\_file = open("text.txt", "r") * # Write to the file * write\_file.write("Not closing files is VERY BAD.") * write\_file.close() * # Try to read from the file * print read\_file.read() * read\_file.close() |
| Not closing files is VERY BAD. |

**The 'with' and 'as' Keywords**

Programming is all about getting the computer to do the work. Is there a way to get Python to automatically close our files for us?

Of course there is. This is Python.

You may not know this, but file objects contain a special pair of built-in methods: \_\_enter\_\_() and \_\_exit\_\_(). The details aren't important, but what *is* important is that when a file object's \_\_exit\_\_()method is invoked, it automatically closes the file. How do we invoke this method? With with and as.

The syntax looks like this:

with open("file", "mode") as variable: # Read or write to the file

**Instructions**

**1.**

Check out the example in the editor. Note that we don't explicitly close() our file, and remember that if we don't close a file, our data will get stuck in the buffer. Click Run!

Success! is written to a file called text.txt.

|  |
| --- |
| with open("text.txt", "w") as textfile:  textfile.write("Success!") |

# Case Closed?

Finally, we'll want a way to test whether a file we've opened is closed. Sometimes we'll have a lot of file objects open, and if we're not careful, they won't all be closed. How can we test this?

f = open("bg.txt") f.closed # False f.close() f.closed # True

Python file objects have a closedattribute which is True when the file is closed and False otherwise.

By checking file\_object.closed, we'll know whether our file is closed and can call close() on it if it's still open.

**Instructions**

**1.**

Below your with...as code, do two things:

* Check if the file is not closed.
* If that's the case, call .close() on it.
* (You don't need an elsehere, since your ifstatement should do nothing if closed is True.)
* After your if statement, print out the value of my\_file.closed to make sure your file is really closed.

Hint

Remember, if statements look like this:

if expression: # Do something

|  |
| --- |
| with open("text.txt","w") as my\_file:  my\_file.write("HELLO")  if my\_file.closed == False:  my\_file.close()  print my\_file.closed |
| True |