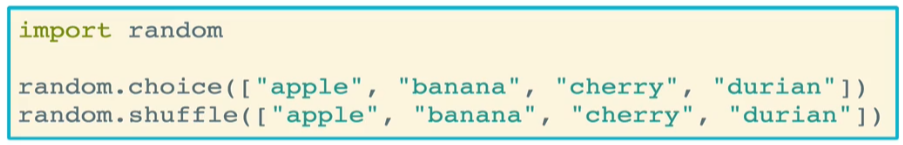
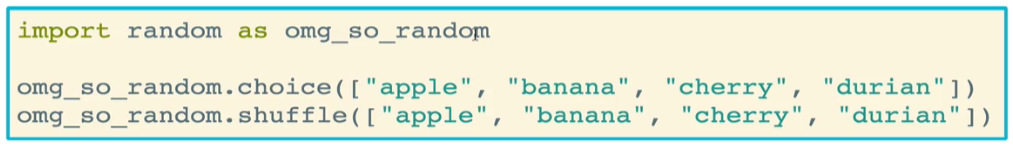
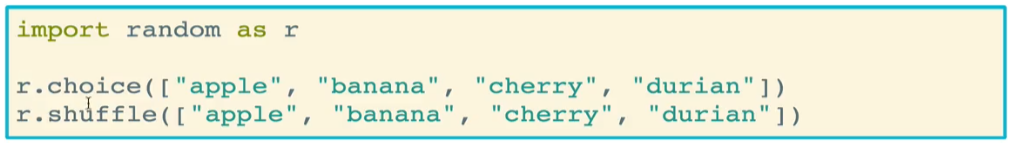
* Modules are, on the surface, a way of writing code in one file and including it into a second file
  + There are many, many pre-written modules that can be included into our Python programs to do both trivial and non-trivial tasks
  + Modules help keep Python files small, similar to how functions help keep the code clean and avoid repetition
  + They also help us re-use code across multiple files using *import*
  + A module is just another Python file. There’s nothing magical about it
* **Built-in modules** are modules that come with Python by default, but must still be manually imported to be used
  + <https://docs.python.org/3/py-modindex.html>
  + Most developers only use a handful of these modules
  + They do not require you to download anything (via pip, for example)
  + Example: the **random** module (<https://docs.python.org/3/library/random.html#module-random>)
    - This module comes with over a dozen different methods that you can use for random stuff
    - The first thing you have to do is import the module
      * If you don’t import, you’ll get a *name error*
    - Then you can access all of the methods within



* + You can also import a module *as* an alias that you can refer to it by, useful when you have long module names (this is a bad example because the alias name is actually longer than the module name)



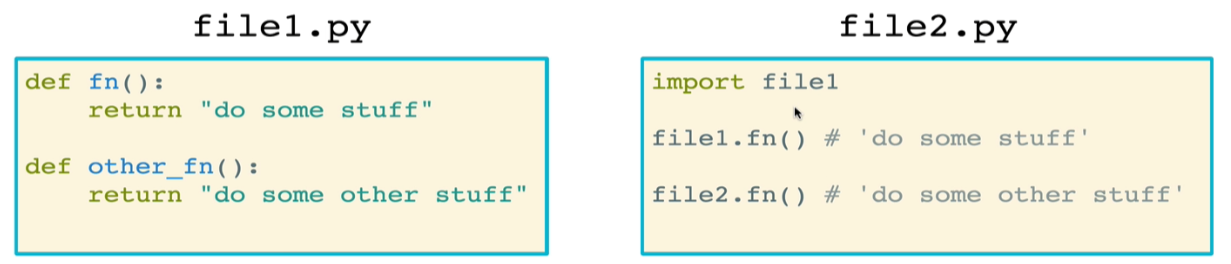
* + - A better example of aliasing



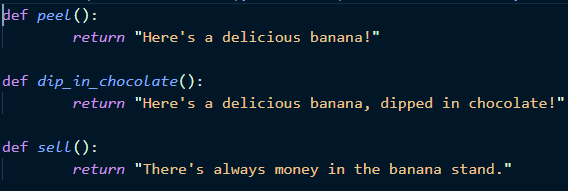
* Using **from** allows you to import only the methods of a module that you need
  + This is much more efficient than importing the entire module, and generally you should only import the methods that are needed for your code
  + Once you have imported the methods, you no longer need to refer to the parent module name. In fact, if you do, you’ll get a NameError
  + You can also import methods using an asterisk. Similar to a normal import statement, it will import all of the methods in the module. However, unlike the normal import statement, the modules will be imported into the *current namespace* as opposed to attributes on the module name



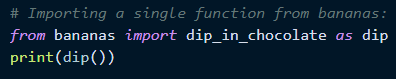
* **Custom Modules** are just files of python code that we can import within other Python files
  + Syntax is the same – import using the name of the file. You can also directly import functions from the file if you so choose
  + If you choose not to import methods directly, then you simply call those methods using the same syntax as we’re familiar with for built-in modules – module\_name.method()
  + Note that custom module files need to be the *same directory* as the file that is calling it. If that is not the case, then you need to reference the correct path to the module file(s)



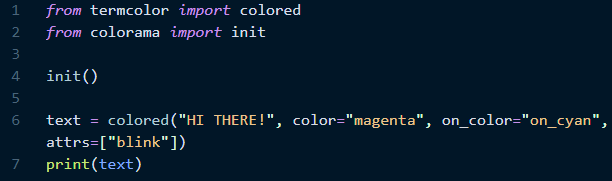
* + Example: fruits
    - The file bananas.py has three functions in it



* + - The file fruits.py can import the bananas file, as well as import the methods within the bananas file directly



* + When should you make a module?
    - When you find yourself with code that needs to be used in more than one place
    - If you have a very long file where you can logically group functions together into separate files
* **External modules** are neither built-in nor custom, but rather are modules built by others that can be downloaded from the internet
  + Python developers all over the world make these modules and we can go download them
  + Many of these are listed on the python package index: <https://pypi.org/>
  + **pip** is the Python management system for Python, and it comes with Python 3 by default
    - Within a python environment, just run **pip install name\_of\_package**
  + Example: termcolor and colorama
    - termcolor is a module used to color text in the terminal
    - colorama is required for Windows systems to use termcolor
    - Code:



* + - Result:



* + Example: **autopep8** is a module that automatically styles your code to match the pep8 style guide
    - <https://pypi.org/project/autopep8/>
    - Oftentimes run in place as there is usually no reason to keep the previous ugly code
    - You can use different options to determine what type of cleanup is performed and how aggressive the cleanup is
* What’s up with the **\_\_name\_\_** variable?
  + Dunder name variable; remember that dunders are not designed to be touched
  + All Python files have a \_\_name\_\_ variable
  + By default, the value of the \_\_name \_\_ variable is simply “\_\_main\_\_”.
    - This is true whenever \_\_name\_\_ is called within the main file being executed
  + Otherwise, if a file is being imported into a different file, then \_\_name\_\_ will take the value of the name of the imported file
  + When you import a module, the following happens:
    - First, Python attempts to find the module (if it fails, it throws an error)
    - Second, it automatically runs the code inside the module being imported. That means if you have any function calls in the imported module (you usually don’t), it will run when imported
      * You can prevent imported code from being run automatically by using a conditional within the module file. When you do this, the imported code will only be run if it is run within its main file, or if it is explicitly called by the file that imported it

