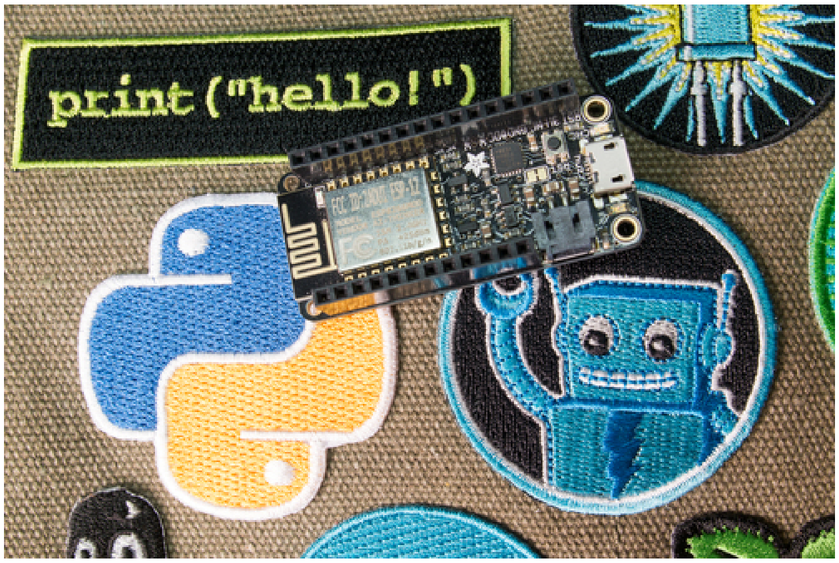
MicroPython Basics: Loading Modules

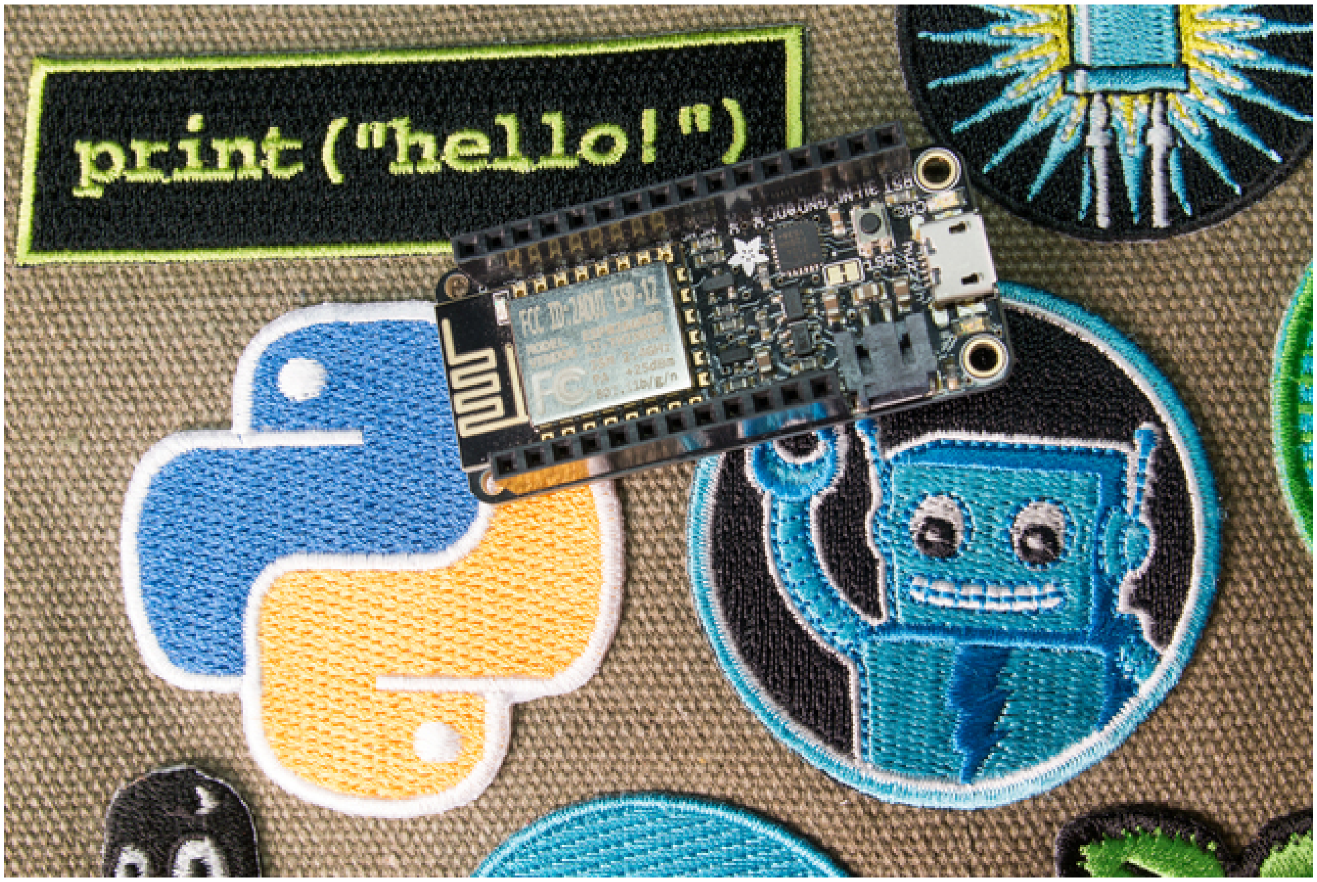
Created by Tony DiCola



Last updated on 2016-08-29 09:26:52 PM UTC

Guide Contents

Overview



Extending your scripts by including code from other files is a great way to simplify and structure programs. Instead of constantly reinventing the wheel for mundane tasks you can put code for them in a Python module or package once and then reuse that code in other scripts. This way you can focus on what's important for your project instead of reimplementing trivial details. Even better by creating modules and packages you can share your code with others so they can benefit from it too!

With MicroPython you can import code in two ways. One easy way is from Python files on a board's filesystem just like you would import Python modules & packages on the desktop. A second way is with 'frozen modules' that are baked-in to a build of MicroPython's firmware and reduce the memory usage of a module vs. importing its raw Python source. This guide explores how to import modules & packages as both raw Python source files and frozen modules for maximum efficiency.

Before you get started you'll want to have a board running MicroPython and be familiar with its basic usage like the serial REPL. In addition for frozen modules you'll need to be familiar with building the MicroPython firmware for your board which is somewhat of an advanced topic. Check out the following guides:

- MicroPython Basics: What is MicroPython (http://adafru.it/pMb)
- MicroPython Basics: How to Load MicroPython on a Board (http://adafru.it/pMf)
- MicroPython Basics: Load Files & Run Code (http://adafru.it/r2B)
- Building and Running MicroPython on the ESP8266 (http://adafru.it/pMB)

Import Code Files

Just like with regular Python you can import and use code from files in your own MicroPython scripts. This is great for breaking a large or complex script into smaller pieces, or for sharing and reusing code with multiple projects.

If you aren't familiar with Python's module support be sure to read the official documentation first (http://adafru.it/rar). Python allows you to put code in a .py file and import it from other scripts in the same directory. You can even get more advanced and create packages which include multiple .py files and expose them in different ways. Most third-party Python libraries are available as packages which you install and import in your own scripts.

We'll start by looking at how to import code from a single **.py** file in your MicroPython script. First make sure you have a board running MicroPython and are familiar with copying files to and from the board.

Next start by creating a simple Python file with a few functions on your computer. In a text editor create **test.py** and fill it with the following code:

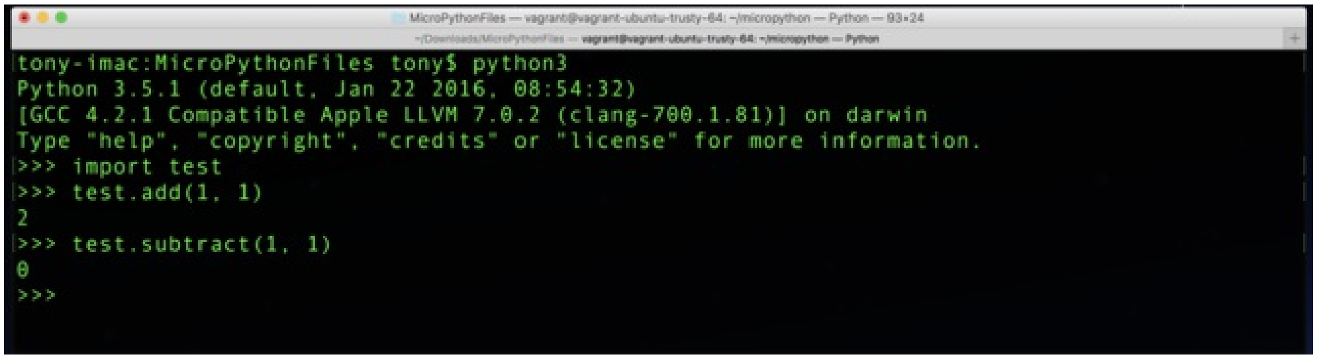
```
def add(a, b):
   return a + b

def subtract(a, b):
   return a - b
```

Try using the code on your computer first with the desktop version of Python (http://adafru.it/cFQ) before trying it in MicroPython. In a terminal navigate to the same directory as the test.py file (this is very important, **you must be in the same directory as test.py!**) and run the **python3** command (or **python** if using Python 2.x). At the Python REPL enter the following commands:

```
import test
test.add(1, 1)
```

You should see the add function called and the result of 1 + 1 returned. If you see an ImportError that the test module can't be found make sure you're running Python from the same directory as **test.py** is located.

Try calling the subtract function just like the add function was called. Remember you need to add the module name in front of the function when you call it!



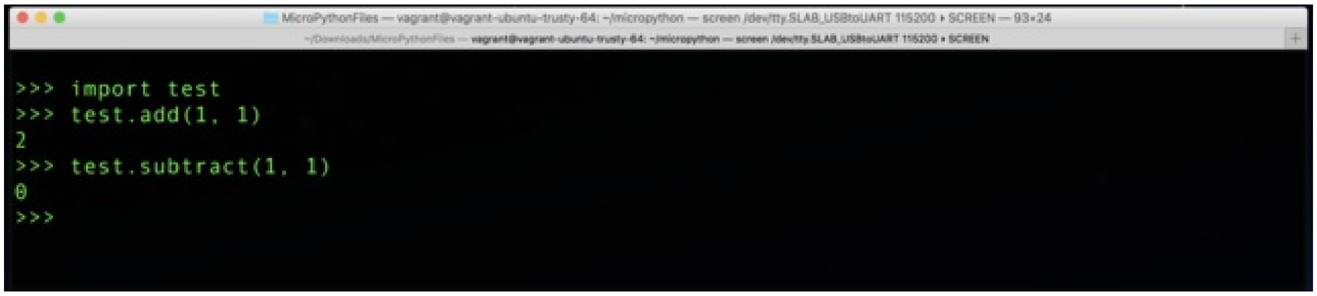
```
tony-imac:MicroPythonFiles tony$ python3
Python 3.5.1 (default, Jan 22 2016, 08:54:32)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import test
>>> test.add(1, 1)
2
>>> test.subtract(1, 1)
0
>>>
```

Now that you see how a simple .py file import works on your computer try doing the same with MicroPython. Copy the test.py file to the root of your board's filesystem. For example if you're using a tool like ampy to copy files you would run something like:

```
ampy --port /board/serial/port put test.py
```

Then connect to the board's REPL and run the same Python code to import and use the module:

```
import test
test.add(1, 1)
test.subtract(1, 1)
```



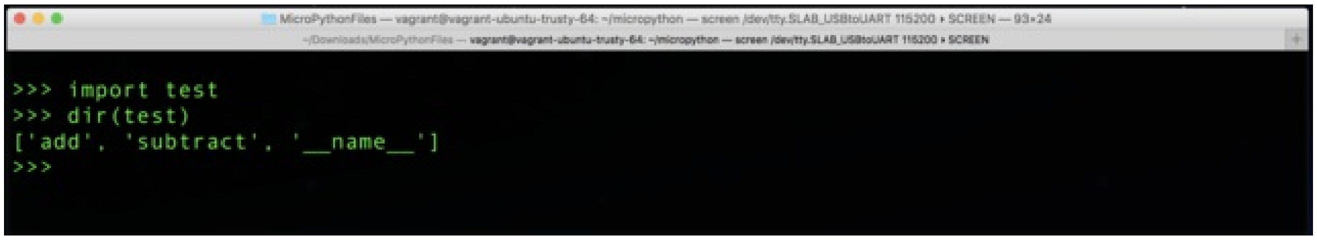
```
>>> import test
>>> test.add(1, 1)
2
>>> test.subtract(1, 1)
0
>>>
```

You should see the functions run just like they did on your computer! If you see an `ImportError` double check you copied the test.py file to the root of the board's filesystem and try again.

Importing and using code from a .py file in MicroPython is as easy as copying the file to the board and importing to use as above. Remember the file needs to be in the same location as the script which is importing and using it. In most cases your scripts will be in the root of the board's filesystem so that's usually where you want to place .py files which will be imported.

You can import and call more than just functions in your scripts too. Anything inside test.py like classes, functions, global variables, etc. will be availabe to your script after the import command runs. In fact you can see exactly what is in the module with the **dir** command, for example in the REPL run:

```
import test
dir(test)
```



```
>>> import test
>>> dir(test)
['add', 'subtract', '__name__']
>>>
```

You should see a list of everything that was imported from the module, including the **add** and **subtract** functions (the **__name__** variable is something Python adds to let the module know what its name is).

Packages

Sometimes your code can get so complex that putting it into a single file doesn't make sense. In these cases you can break code into multiple files and create a Python package that puts all the code together into what looks like a simple module your scripts can import. MicroPython supports the concept of Python packages just like normal Python so you can better structure complex scripts.

First be sure to read the official documentation on Python packages (http://adafru.it/rar). Packages in MicroPython for the most part work just the same as in Python.

Now create a Python package on your computer by creating a directory called **test**. Inside that directory create a file called **add.py** and place inside it the add function code:

```
def add(a, b):
   return a + b
```

Create a file **subtract.py** in the same location and place in it the subtract function code:

```
def subtract(a, b):
   return a - b
```

Finally create a file called **__init__.py** in the same location and enter the following code to import and expose the functions from the files above:

```
from test.add import add
from test.subtract import subtract
```

Be sure to call this file exactly __init__.py! Python looks for this file name to know that it found a package. If you don't have this file or if it's not named correctly then Python will fail to import the package!

The code inside **__init__.py** runs when the package is imported in a script. You can see this code imports the add function from the **add.py** script in the package, and the subtract function from the **subtract.py** script.

Notice the import statements refer to**add.py** by its full 'absolute import' name of**test.add**. The test. in front of add is the package name, i.e. the name of the package directory. You can't run a command like '**from add import add**' in the __init__.py since Python would get confused if it should load an add.py from inside the module or from elsewhere. Using the full name **test.add** tells Python to use the**add.py** module inside the **test** package directory.

Double check you have a test package structure that looks like the following:

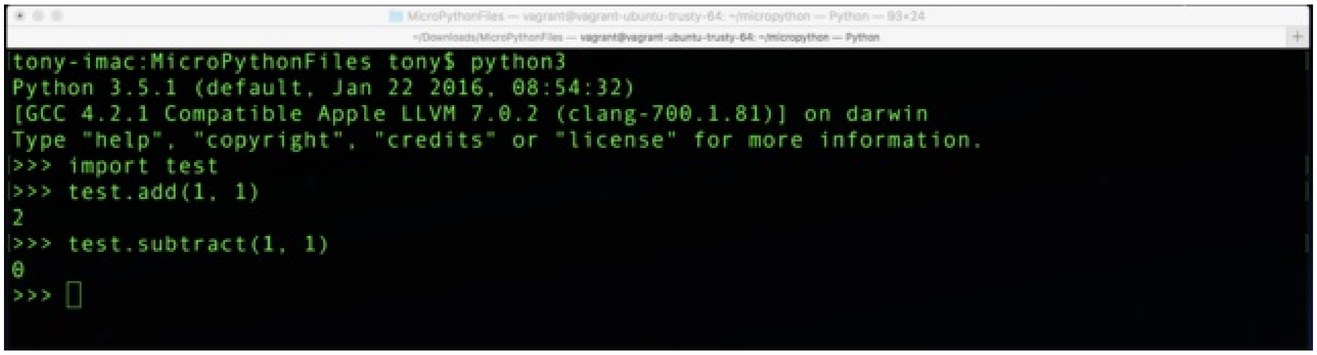
- **test** directory
  - **__init__.py** file which imports the add and subtract functions
  - **add.py** which exposes the add function
  - **subtract.py** which exposes the subtract function

Now try importing and using the package with desktop Python. In a terminal navigate to the **parent of the test package directory** (i.e. one folder above it). Importing and using a package is just like importing a .py file however Python treats the entire test directory as the package itself. This means you need to run Python from above the test directory so Python can find the package.

Also note be sure you don't have a test.py file in the same directory as the test package! If you do Python could get confused and import the test.py file instead of the test package. Delete test.py if it exists next to the test package folder!

Run the following code in the Python REPL to import and use the add and subtract functions from the test package:

```
import test
test.add(1, 1)
test.subtract(1, 1)
```


MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — Python — 93×24
~/Downloads/MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — Python

```
tony-imac:MicroPythonFiles tony$ python3
Python 3.5.1 (default, Jan 22 2016, 08:54:32)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import test
>>> test.add(1, 1)
2
>>> test.subtract(1, 1)
0
>>> 
```

You should see the package imported and the functions work exactly as you saw before! If you see an ImportError that the test module doesn't exist be sure you're running Python from the **parent** of the test package directory!

Now try copying the package to your MicroPython board and using it exactly as you did with desktop Python. **Again make sure a test.py file doesn't exist on the board or else MicroPython will be confused about what to import!** If you have test.py on your board already you can delete it using the following ampy command:

```
ampy --port /board/serial/port rm test.py
```

Now create the test package directory in the root of the board:

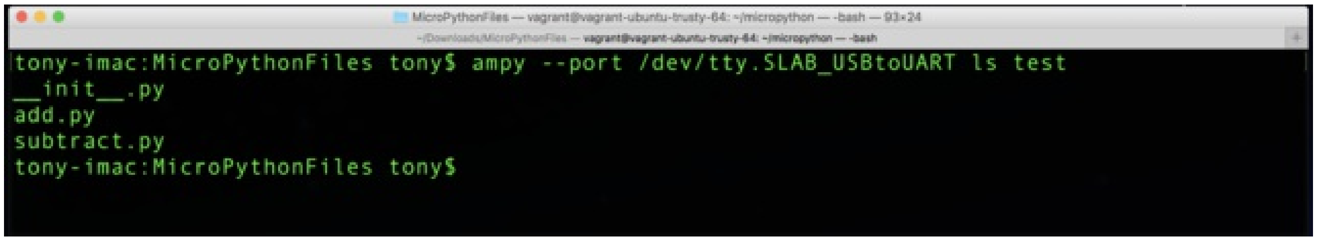
```
ampy --port /board/serial/port mkdir test
```

And copy inside the three .py files above that define the package (run these commands from **inside** the test package folder):

```
ampy --port /board/serial/port put __init__.py test/__init__.py
ampy --port /board/serial/port put add.py test/add.py
ampy --port /board/serial/port put subtract.py test/subtract.py
```

These commands will copy the .py files to the test folder on the board's filesystem. Use the ls command to double check there's a test folder with these three files on your board:

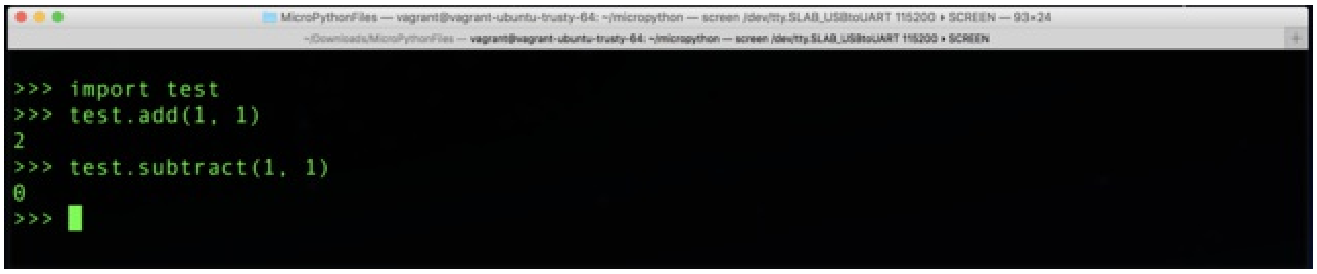
```
ampy --port /board/serial/port ls test
```


MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — -bash — 93×24
~/Downloads/MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — -bash

```
tony-imac:MicroPythonFiles tony$ ampy --port /dev/tty.SLAB_USBtoUART ls test
__init__.py
add.py
subtract.py
tony-imac:MicroPythonFiles tony$
```

Connect to the board's REPL and try importing the test package and using its functions as before:

```
import test
test.add(1, 1)
test.subtract(1, 1)
```


MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — screen /dev/tty.SLAB_USBtoUART 115200 • SCREEN — 93×24
~/Downloads/MicroPythonFiles — vagrant@vagrant-ubuntu-trusty-64: ~/micropython — screen /dev/tty.SLAB_USBtoUART 115200 • SCREEN

```
>>> import test
>>> test.add(1, 1)
2
>>> test.subtract(1, 1)
0
>>> 
```

You should see the package import and the functions work exactly as before when they

You should see the package import and the functions work exactly as before when they were in a single test.py file!

Breaking a complex module apart into multiple files with a package is a great way to simplify and structure code. In this simple example it seems like a bit of unnecessary work, but as scripts get more complex and re-use common code it will help immensely to break them into modules and packages. You can even start to share code with others by giving them your modules and packages to load and use!

Frozen Modules

In the previous page you saw how to load modules and packages from Python code files on a board's filesystem. This is a quick and easy way to load modules, but there are some big limitations with storing and running code from the filesystem. The biggest limitation is that raw Python code has to be loaded and processed by MicroPython's interpreter. This process takes some time and memory which on some boards is quite limited. You might even find that some code files are so large they can't be loaded in memory and processed by MicroPython's interpreter!

However there is a way to pre-process modules so they're 'baked-in' to MicroPython's firmware and use less memory. MicroPython can squeeze almost all of the Python language into kilobytes of memory by 'freezing' Python code so that it's stored as more efficient byte-code or even compiled into native code instead of being stored as raw Python code. Once the code is frozen it can be quickly loaded and interpreted by MicroPython without as much memory and processing time.

With MicroPython you can freeze your own modules and add them to a custom build of the

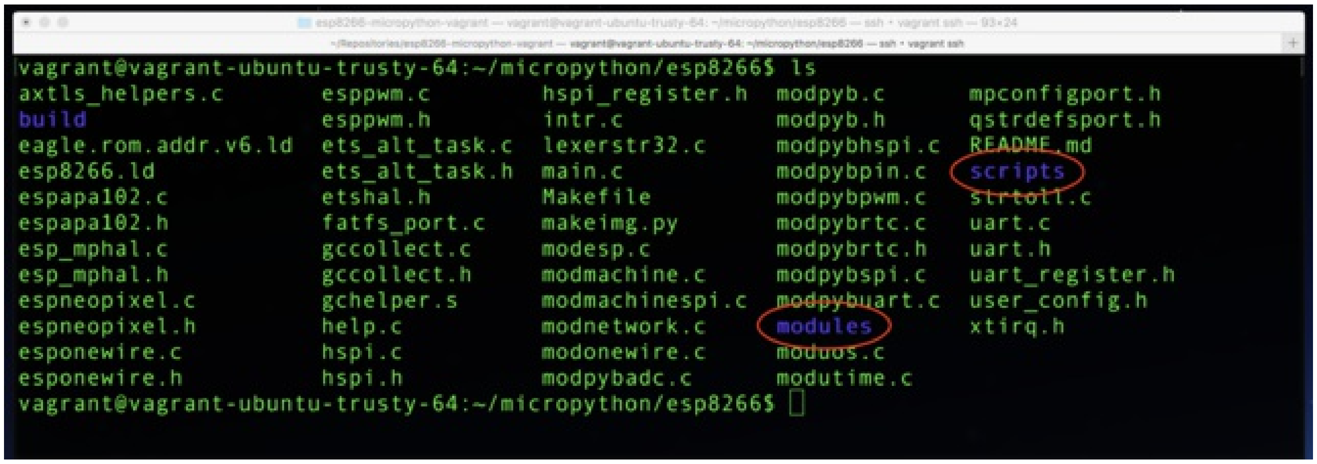
MicroPython firmware for your board. This allows you to write more complex modules than you might be able to run directly from .py Python code files as previously shown.

Be aware that freezing modules is a bit advanced and requires you to make a custom build of MicroPython's firmware. For some boards like the ESP8266 it's relatively easy to build MicroPython firmware, but for others it might require more advanced knowledge of setting up and using compiler toolchains, etc.

Before you look at freezing modules make sure you can compile MicroPython firmware for your board. For the ESP8266 check out this handy guide on how to compile MicroPython firmware (http://adafru.it/pMB) using a special Linux-based virtual machine. For other boards like the pyboard, WiPy, etc. for now you'll need to check the MicroPython source (http://adafru.it/fa3) to see how to setup their toolchains and compile firmware.

Also be aware freezing modules is somewhat new for boards like the ESP8266 and the process might change over time. If the steps below have issues, check the latest documentation, code on GitHub, and MicroPython forums to see if more up to date information on freezing modules is availble.

For this guide we'll look specifically at the ESP8266 and how to add a frozen module to the board's MicroPython firmware. Start by looking at the source of the MicroPython ESP8266 firmware. If you're using the Vagrant virtual machine from the previously mentioned firmware building guide (http://adafru.it/pMB) then you'll want to enter the VM with SSH and navigate to the ~/**micropython/esp8266** folder. Notice there are two subdirectories, **scripts** and **modules**:



```
vagrant@vagrant-ubuntu-trusty-64:~/micropython/esp8266$ ls
axtls_helpers.c       esppwm.c        hspi_register.h  modpyb.c       mpconfigport.h
build                 esppwm.h        intr.c           modpyb.h       qstrdefsport.h
eagle.rom.addr.v6.ld  ets_alt_task.c  lexerstr32.c     modpybhspi.c   README.md
esp8266.ld            ets_alt_task.h  main.c           modpybpin.c    scripts
espapa102.c           etshal.h        Makefile         modpybpwm.c    strtoll.c
espapa102.h           fatfs_port.c    makeimg.py       modpybrtc.c    uart.c
esp_mphal.c           gccollect.c     modesp.c         modpybrtc.h    uart.h
esp_mphal.h           gccollect.h     modmachine.c     modpybspi.c    uart_register.h
espneopixel.c         gchelper.s      modmachinespi.c  modpybuart.c   user_config.h
espneopixel.h         help.c          modnetwork.c     modules        xtirq.h
esponewire.c          hspi.c          modonewire.c     moduos.c
esponewire.h          hspi.h          modpybadc.c      modutime.c
vagrant@vagrant-ubuntu-trusty-64:~/micropython/esp8266$ 
```

There's an important difference between these two folders. The**scripts** folder is for Python code that will be 'baked-in' to the firmware but **not frozen** into more efficient and smaller code. Everything in the scripts folder is just stored as the raw Python code in the board's flash memory. This saves you from having to copy that code onto the board's filesystem, but doesn't save a lot of memory or processing time.

The **modules** folder is for Python code that will be frozen into more efficient bytecode. This is where you want to place scripts that you'd like freeze to save memory.

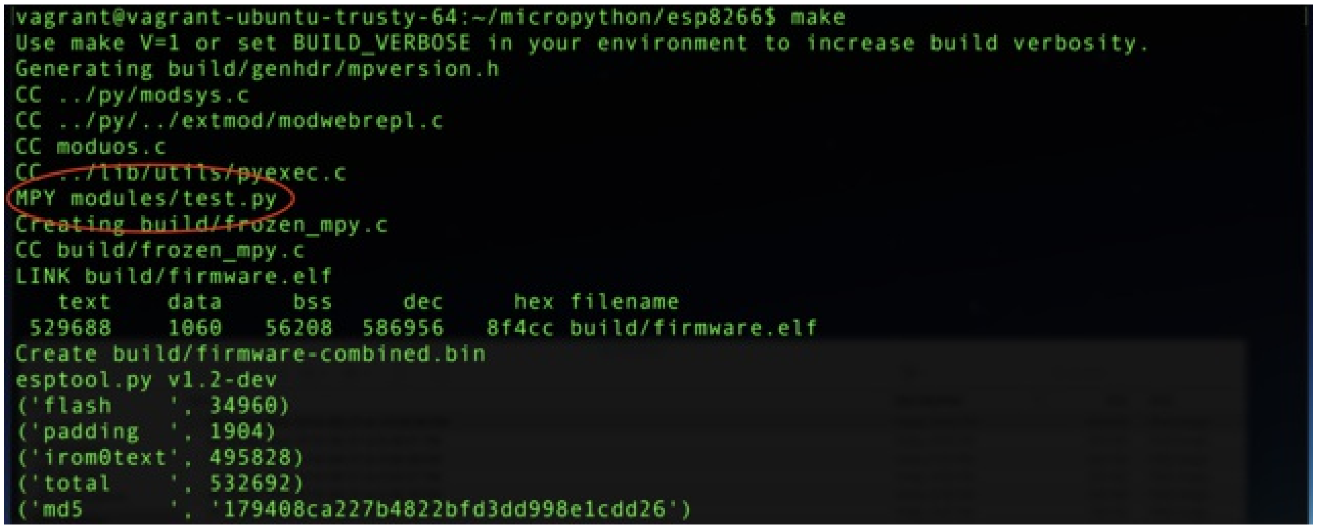
Try adding a new frozen module as a quick test. Inside the **modules** folder create a **test.py** file and add the following code to it:

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

Now rebuild the ESP8266 MicroPython firmware. From inside the VM's **~/micropython/esp8266** folder run:

```
make
```

Notice in the output the **MPY modules/test.py** line, this shows the **test.py** module was picked up by the build and frozen into bytecode:



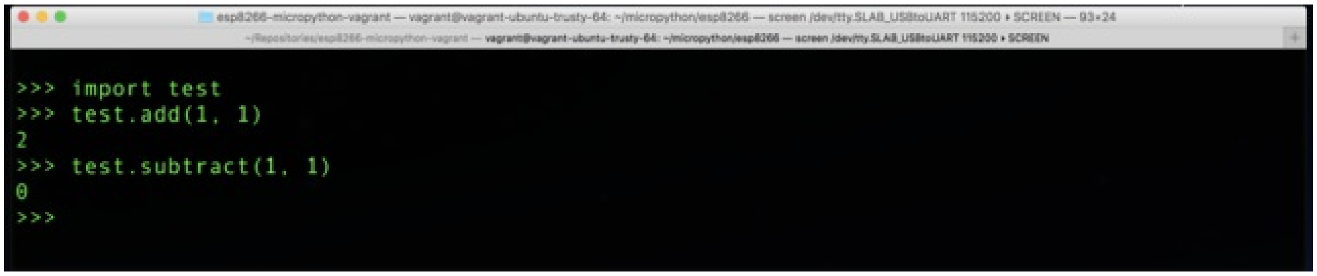
```
vagrant@vagrant-ubuntu-trusty-64:~/micropython/esp8266$ make
Use make V=1 or set BUILD_VERBOSE in your environment to increase build verbosity.
Generating build/genhdr/mpversion.h
CC ../py/modsys.c
CC ../py/../extmod/modwebrepl.c
CC moduos.c
CC ../lib/utils/pyexec.c
MPY modules/test.py
Creating build/frozen_mpy.c
CC build/frozen_mpy.c
LINK build/firmware.elf
   text    data     bss     dec     hex filename
 529688    1060   56208  586956   8f4cc build/firmware.elf
Create build/firmware-combined.bin
esptool.py v1.2-dev
('flash    ', 34960)
('padding  ', 1904)
('irom0text', 495828)
('total    ', 532692)
('md5      ', '179408ca227b4822bfd3dd998e1cdd26')
```

If you don't see this line for your module then try running the **make clean** command and then **make** command again.

Now copy out the newly built firmware and flash it to the ESP8266. Remember you're now using a development build so debug output will be enabled and features like the WebREPL must be manually started.

Connect to the REPL of the board with the new firmware and try using the test module's functions by running:

```
import test
test.add(1, 1)
test.subtract(1, 1)
```



```
>>> import test
>>> test.add(1, 1)
2
>>> test.subtract(1, 1)
0
>>>
```

You should see the module import and functions work just like if the .py file was on the filesystem!

You can also freeze Python packages inside the **modules** folder. Just create a directory for each package and put inside it an **__init__.py** and other source files for the package.

Using the **scripts** folder to save Python module sources in flash memory is just like using the modules folder above. However note with the scripts folder you cannot put packages inside it, only single .py source files will currently work.

Remember the **scripts** folder is only for scripts that will be saved as-is to flash memory. The **modules** folder is for freezing modules and packages so they're in flash memory and more efficient to load and store. As you develop your own complex MicroPython scripts and modules consider freezing them in the MicroPython firmware to reduce memory usage and increase performance!