Modules

Single Py Files Containing	Python components you have defined (functions, variables, classes, etc.) Runnable code (scripts)
Can be:	Executed (python my_module.py) Imported from a shell or another file
IMPORTING	
When you import a module, a new name, 'bound' to the module, is created in the current scope:	import math x = math.pow(2, 3)
	from math import pow x = pow(2, 3)
We can also import a specific function from the module, and so now we don't need to specify the module name	from math import * x = pow(2, 3)
We could also achieve this by importing all objects in the module, but isk creating namespace conflicts:	from math import * x = pow(2, 3)
We can import a module under an alias to bind it to a name of our choice:	import pandas as pd import seaborn as sns

All runnable code in a module is executed at import.

Modules (cont)

f you want it to be executed only when you run the file, use the following:

if __name__ == "__main__": print("This will be run only if the file is execut ed")	__name__ is a special built-in variable which will automatically be set to "__main__" if the source file is being executed as the main program, rather than being imported.

When importing, Python looks for the module with the same name in:

The built-in modules	The directories defined in the sys.path list:
	The current working directory The PYTHONPATH list The default Python directory
You can access this list at runtime and append new paths to it manually:	import sys print(sys.path) sys.path.append("/path/to/add")

Code to treat Jupyter notebooks as modules

```
import io, os, sys, types
import nbformat
from IPython import get_ipython
from IPython.core.interactiveshell import InteractiveShell
def find_notebook(fullname, path=None):
"""find a notebook, given its fully qualified name and an optional path
This turns "foo.bar" into "foo/bar.ipynb"
```

Code to treat Jupyter notebooks as modules (cont)

```
and tries turning "Foo_Bar" into "Foo Bar" if Foo_Bar
does not exist.
"""
name = fullname.rsplit('.', 1)[-1]
if not path:
path = ['']
for d in path:
nb_path = os.path.join(d, name + ".ipynb")
if os.path.isfile(nb_path):
return nb_path
# let import Notebook_Name find "Notebook Name.ipynb"
nb_path = nb_path.replace("_", " ")
if os.path.isfile(nb_path):
return nb_path
class NotebookLoader(object):
"""Module Loader for IPython Notebooks"""
def __init__(self, path=None):
self.shell = InteractiveShell.instance()
self.path = path
def load_module(self, fullname):
"""import a notebook as a module"""
path = find_notebook(fullname, self.path)
print ("importing notebook from %s" % path)
# load the notebook object
nb = nbformat.read(path, as_version=4)
# create the module and add it to sys.modules
# if name in sys.modules:
# return sys.modules[name]
mod = types.ModuleType(fullname)
mod.__file__ = path
mod.__loader__ = self
mod.__dict__['get_ipython'] = get_ipython
sys.modules[fullname] = mod
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Code to treat Jupyter notebooks as modules (cont)

```
# extra work to ensure that magics that would affect the user_ns
# actually affect the notebook module's ns
save_user_ns = self.shell.user_ns
self.shell.user_ns = mod.__dict__
try:
for cell in nb.cells:
if cell.cell_type == 'code':
# transform the input to executable Python
code = self.shell.input_transformer_manager.transform_cell(cell.source)
# run the code in themodule
exec(code, mod.__dict__)
finally:
self.shell.user_ns = save_user_ns
return mod
class NotebookFinder(object):
"""Module finder that locates IPython Notebooks"""
def __init__(self):
self.loaders = {}
def find_module(self, fullname, path=None):
nb_path = find_notebook(fullname, path)
if not nb_path:
return
key = path
if path:
# lists aren't hashable
key = os.path.sep.join(path)
if key not in self.loaders:
self.loaders[key] = NotebookLoader(path)
return self.loaders[key]
sys.meta_path.append(NotebookFinder())
```

Importing ipynb as modules

from NameOfModule import NameOfFunction	from math import pow2

Packages

Packages are directories that contain modules and/or other packages. This can be a good way to group modules in a hierarchical directory structure.

__init__.py in a package will be executed at import.	Can be used to import nested modules at a higher level of hierarchy

When working with packages, it is recommended to assume code will be run from the top level and use absolute imports from there

Add an extra level of hierarchy and a setup.py file	To install run: pip install my_package/ (the / is important)

Once installed, you can import your module from Python, or run an executable in the shell if you've defined an entrypoint.

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com