

## What is a Python Module?

### Python Modules

- A module is a Python file ending in .py
- Groups related functions, classes and metadata
- Python Naming Convention (PEP8)
  - lower\_case\_name\_with\_underscores
- A python package is composed of several modules
  - A python package is a folder containing **init.py**
  - Groups modules with related or complimentary concepts

### Import Statement

- Pulls the functions in an imported module into the globals() namespace
  - These become accessible 'globally' within that Python process
  - Alias given at import time only accessible within the calling module
  - ie: 'import arcpy'
    - \* Can type arcpy.member\_name only within the importing module
    - \* If imported in a second module, there is no load time
- Module Objects
  - Reference objects to the imported module's collection of function byte-code
  - Define what is visible to consumers via **all** = [] list of strings
  - Used by most 'intellisense' style auto-complete
  - Currently arcpy.da does NOT have an **all** therefore it does not autocomplete

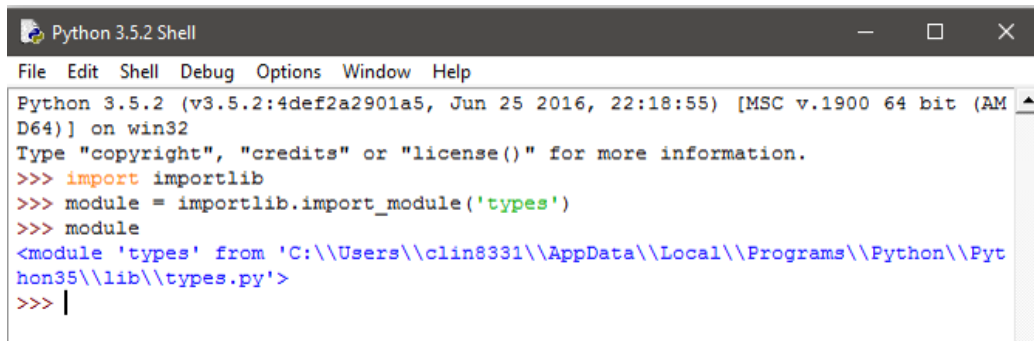
### importlib

- Introduced in Python 3.1

Libraries > Pro > Resources > ArcPy > arcpy >	
Name	Date modified
__pycache__	3/8/2017 12:0
arcobjects	3/8/2017 12:0
geoprocessing	3/8/2017 12:0
sa	3/8/2017 12:0
__init__.py	12/8/2016 10:
_base.py	8/27/2014 10:
_chart.py	12/7/2016 12:
_ga.py	2/26/2016 3:4
_graph.py	8/23/2016 2:2
_import_list.py	8/27/2014 10:
_importable_modules.py	8/27/2014 10:
_management.py	8/27/2014 10:
_mp.py	12/6/2016 3:5
_na.py	9/25/2016 9:5
_renderer.py	12/7/2016 8:4
_symbology.py	12/7/2016 8:4
_wmx.py	2/21/2017 6:2
analysis.py	1/19/2017 5:0
cartography.py	1/26/2017 12:
conversion.py	1/30/2017 1:4
da.py	8/27/2014 10:
ddd.py	2/14/2017 2:3
edit.py	1/19/2017 5:0
ga.py	1/19/2017 5:0
geoanalytics.py	2/14/2017 2:3

Figure 1:

- Deprecates the 'imp' module
- `Importlib.import_module('module_name')` will return a module object
  - Explore modules by assigning the module object and looking at its `dir()`
  - `dir()` lists all members in a module (including `_private` members)
    - \* Functions/Classes/Variables with a leading `_`
    - \* Not imported with 'import \*' – unless added explicitly to **all**



```

Python 3.5.2 Shell
File Edit Shell Debug Options Window Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import importlib
>>> module = importlib.import_module('types')
>>> module
<module 'types' from 'C:\Users\clin8331\AppData\Local\Programs\Python\Python35\lib\types.py'>
>>> |
  
```

Figure 2:

## Importing Best Practices

- Break functionality into small modules, import the finest grain of functionality
- Control loading of multiple modules via `__init__.py` in a Package
- Do not use 'from library import \*' in modules designed to be consumed
  - May cause namespace collisions
    - \* Two members from different modules with the same name
  - Avoid by importing explicitly
    - \* 'from library import member\_name'
  - Correct namespace collisions using `as` keyword
    - \* 'from library import member\_name as unique\_member\_name'

## Importing Best Practices

- Use `importlib` to access module objects
  - Useful for 'lazy loading' of specific modules
  - Also for conditional loading of specific modules

- Liberal use of `_private` variable names
  - Hides most unwanted access to members
  - Impossible to completely hide a member in Python
- Concise import statements
  - Be mindful of namespace collisions both down and upstream

## How do Modules help?

### Code Reuse

- Modular code can be consumed in many places
  - Write and test code in one place
  - Avoid duplicating functions across tools
- Regularly refactor code to facilitate reuse
  - View code as layers of functionality
    - \* Data Model
    - \* Functional Model
    - \* View Model
    - \* View
- Decreases time, money and effort spent in the future
  - Ease of use
  - ‘Self Documenting’
  - Don’t reinvent the wheel

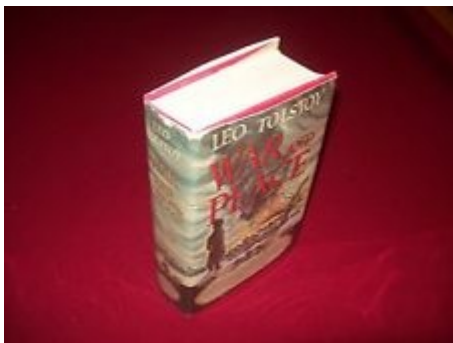
### Testability

- Testing smaller increments of functionality in modules
  - Focused testing
  - Limited exposure to side-effects from other code
  - Easy-to-follow debugging experience
- ‘Unit Tests’
  - The smallest units of code within the module
  - Should only have one job with defined edges

- \* 'edge cases' aka functional boundaries
- 'Integration Tests'
  - Interoperation between each function in multiple modules
  - Harder to identify bugs
  - More involved debugging process

## Interchangeability

- Smaller units of code are easier to replace
  - Improved method
  - Updating variable names
- Also easier to understand
  - Reading 'War and Peace' vs. Reading several small books
- Quickly prototype changes
  - Create a test version with a module replaced
- Similar to building something with Lego vs. Concrete



## Multiple 'Views'

- Functionality like `arcpy.GetParameter()` in separate file from logic
- Consume modularized ArcPy code:
  - .tbx
  - .pyt
  - Python Window

- Command Line
- Only write the tool one time
- Toolboxes provide different validation functionality
  - .tbx through the Geoprocessing pane in application
  - .pyt through python code in the toolbox file

**Thank you!! Questions/Comments?? [cdow@esri.com](mailto:cdow@esri.com)**