* Course Overview
* Prerequisites
  + Imported nested packages
  + Packages are modules
  + Locating modules for import
  + Modularization
    - A -> B -> C
  + Prerequisites
  + Main Block()
    - Differentiate Module execution from module import
  + \_\_<method-name>\_\_
    - Special methods
    - Dunder(double underscore) method name
* Introduction to Packages
  + Modules
    - Python’s basic tool for organizing code
    - Normally a single python source file
    - Load modules with import
    - Represented by module objects
  + Packages: Modules that contain other modules
  + urllib is a package
    - has a \_\_path\_\_ member
  + urllib.request is a module
    - doesn’t have a \_\_path\_\_ member
  + packages are generally directories
  + modules are generally files
* Locating Modules
  + Python looks on filesystem for corresponding python file
  + Python use sys.path
    - List of directories
    - Then searched in order in import
    - First match provides module
    - ImportError when there is no match
  + sys.path[0] is empty
    - when you start interpreter with no arguments
  + can add places for python to search
  + ex)
    - import sys
    - sys.path.append(‘not\_searched’)
  + PYTHONPATH
    - Environment variable
    - Lists of paths added to sys.path when python starts
    - Windows: set PYTHONPATH=path1;path2;path3
    - Linux/macOS: export PYTHONPATH=path1:path2:path3
* Summary
  + Importing nested packages
  + All modules in hierarchy are imported
    - Only the first name is bound
    - Use fully-qualified names for submodules
  + Package directory paths are stored in \_\_path\_\_
  + Sys.path contains module search
    - It is initialized from PYTHONPATH
* Creating Packages
  + create directory in sys.path
  + create a file called ‘\_\_init\_\_.py’
    - its what make the package a module
    - often empty
    - optional in python 3.3+
    - still required in earlier python versions
    - powerful initialization tool
    - explicit is better than implicit
  + module.\_\_file\_\_
    - returns path to \_\_init\_\_.py file
  + a package is a directory container \_\_init\_\_.py
* Creating a Subpackage
  + create a directory within a directory
  + the sub directory must also include \_\_init\_\_.py
  + import gzip to work with compressed file
  + gzip.open
    - decompresses during reading
  + python -m [full path module name] [sys.argv[1]] [sys.argv[2]...sys.argv[5]
    - -m: run module flag
  + ex)
    - python -m demo\_reader.compressed.bzipped test.bz2 data compressed with bz2
* Relative Imports
  + Absolute Imports
    - specify all ancestors modules of modules you want to import
  + ex)
    - import demo\_reader.compressed.bzipped
    - from demo\_reader.compressed import bzipped
  + Relative Import Syntax
    - use shortened paths to modules and packages
  + ex)
    - from ..module\_name import name
  + each dot before module\_name stands for an ancestor package of module doing the import
  + you can only use relative imports with ‘from module import name’ form of import
  + relative imports can only be used to import modules within the current top-level package
  + can reduce typing in deeply nested package structures
  + promote a certain form of modifiability
  + in general prefer absolute import
* Using \_\_all\_\_
  + module-level attribute
  + controls from module import \* behavior
  + if not specified, imports all public names
  + must be a list of string
    - each entry is a name to import
  + ex in \_\_init\_\_.py
    - from demo\_reader.compressed.bzipped import opener as bz2\_opener
    - from demo\_reader.compressed.gzipped import opener as gzip\_opener
    - \_\_all\_\_ = [‘bz2\_opener’, ‘gzip\_opener’]
  + while \_\_all\_\_ can be useful
    - we recommend avoiding import \* in general
  + packages are modules which can container other modules
  + directories containing \_\_init\_\_.py
  + technically optional
    - but its presence is an explicit signal to developers
    - executed at package import
  + packages can contain subpackages
  + \_\_all\_\_ controls import \* behavior
* Namespace Packages
  + May want to split packages across directories
  + PEP 420: Implicit Namespace Packages
    - Namespace packages are a mechanism for splitting a single python package across multiple directories on disk
  + Namespace packages may not have \_\_init\_\_.py
  + Namespace package discovery algorithm
    - Scans each directory in sys.path in order
    - Import standard package if found
    - Import standard module if found
    - Otherwise, all matching directories contribute to a namespace package
  + Ex)
    - Folder path1
      * Folder demo\_reader
        + Folder Util

File \_\_init\_\_.py, write.py

* + - * + File multireader.py
    - Folder path2
      * Folder demo\_reader
        + Folder compressed

File \_\_init.py, bzipped.py, gzipped.py

* + Add path1 and path2 to sys.path
  + Ex in REPL)
    - import sys
    - sys.path.extend([‘./path1’, ‘./path2’])
* Executable Directories
  + Let you specify a main entry point which is run when the directory is executed by python
  + Have a directory with no \_\_init\_\_.py files
    - Have it contain the packages you want to run
    - Have it contain a \_\_main\_\_.py file
  + You can execute a directory if it contains \_\_main\_\_.py
    - \_\_main\_\_.py will be executed
  + In in command line
    - python multi-reader-program
  + \_\_main\_\_.py is added to sys.path
    - Then its executed by ‘python directory’
* Executable Zip Files
  + Python knows how to read zip files and treat them like directories
  + Create zip file from your directory
    - Zip file contains the same contents as the directory, no the directory itself
  + Then in command line
    - python multi-reader-program.zip test.bz2
* Executable Packages
  + Add \_\_main\_\_.py in packages them make them executable
    - Executed by ‘python -m package’
  + Executing Directories vs Packages
  + python directory
    - executing a directory
    - “directory” added to sys.path
    - “directory/\_\_main\_\_.py” is not in a package
    - sys.path contains demo reader itself not the directory
  + python -m directory
    - executing a package
    - the “-m” tells python to treat it as a module
    - “directory” treated as a package
    - “directory/\_\_main\_\_.py” is a submodule of the directory package
  + \_\_init\_\_.py vs \_\_main\_\_.py
    - \_\_init\_\_.py can execute any code it likes on import
    - Only a package with \_\_main\_\_.py can be executed
  + Construct packages from multiple directories with namespace packages
  + Namespace packages cannot contain \_\_init\_\_.py
  + Directories can be made executable with \_\_main\_\_.py
  + Python can execute zip files like directories
  + Packages can be both importable and executable with \_\_main\_\_.py
* Python Project Structure
  + Ex) / is folder
    - Project\_name/
    - README.rst #overview, reStructuredText
    - docs/ #project documentation, easy to find, should be in project root
    - src/ #package/production code
    - package\_name/
    - \_\_init\_\_.py
    - more\_source.py
    - subpackage1/
    - \_\_init\_\_.py
    - tests/ #all tests for the project
    - test\_code.py
    - setup.py
  + Src directory ensures that you develop against installed versions of your packages
  + Separation of test and production code
    - Test and production code server different purposes
    - Usually don’t want tests installed with package
    - Avoid tool treating tests as production code
  + Be pragmatic! Put tests in production code if necessary
* A concrete Example: demo\_re..