Programming Principles in Python (CSCI 503)

Arrays

Dr. David Koop
Modules and Packages

- Python allows you to import code from other files, even your own
- A **module** is a collection of definitions
- A **package** is an organized collection of modules
- Modules can be
  - a separate python file
  - a separate C library that is written to be used with Python
  - a built-in module contained in the interpreter
  - a module installed by the user (via conda or pip)
- All types use the same import syntax
What is the purpose of having modules or packages?

- Code reuse: makes life easier because others have written solutions to various problems
- Generally forces an organization of code that works together
- Standardizes interfaces; easier maintenance
- Encourages robustness, testing code

- This does take time so don't always create a module or package
  - If you're going to use a method once, it's not worth putting it in a module
  - If you're using the same methods over and over in (especially in different projects), a module or package makes sense
Importing modules

- `import <module>`
- `import <module> as <another-identifier>`
- `from <module> import <identifier-list>`
- `from <module> import <identifier> as <another-identifier>, ...`

- `import imports from the top, from ... import imports "inner" names`
- Need to use the qualified names when using import (`foo.bar.mymethod`)
- `as clause renames` the imported name
Namespaces

• Namespace is basically a dictionary with names and their values
• Accessing namespaces
  - __builtins__, globals(), locals()
• Examine contents of a namespace:
  - dir(<namespace>)
• Python checks for a name in the sequence:
  local, enclosing, global, builtins
• To access names in outer scopes, use
  global (global) and nonlocal (enclosing) declarations
Using an imported module

• Import module, and call functions with **fully qualified** name
  - import math
    math.log10(100)
    math.sqrt(196)

• Import module into current namespace and use **unqualified** name
  - from math import log10, sqrt
    log10(100)
    sqrt(196)
Reloading a Module?

• If you re-import a module, what happens?
  - import my_module
    my_module.SECRET_NUMBER # 42
  - Change the definition of SECRET_NUMBER to 14
    - import my_module
      my_module.SECRET_NUMBER # Still 42!
  - Modules are **cached** so they are not reloaded on each import call
  - Can reload a module via `importlib.reload(<module>)`
  - Be careful because **dependencies** will persist! (Order matters)
Python Packages

• A package is basically a collection of modules in a directory subtree
• Structures a module namespace by allowing dotted names
• Example:
  ```python
  - test_pkg/
    __init__.py
    foo.py
    bar.py
    baz/
      fun.py
  ```
• For packages that are to be executed as scripts, `__main__.py` can also be added
Finding Packages

- Python Package Index (PyPI) is the standard repository (https://pypi.org) and pip (pip installs packages) is the official python package installer
  - Types of distribution: source (sdist) and wheels (binaries)
  - Each package can specify dependencies
  - Creating a PyPI package requires adding some metadata
- Anaconda is a package index, conda is a package manager
  - conda is language-agnostic (not only Python)
  - solves dependencies
  - conda deals with non-Python dependencies
  - has different channels: default, conda-forge (community-led)
Installing Packages

• `pip install <package-name>`
• `conda install <package-name>`
• Arguments can be multiple packages
• Be careful! Security exploits using package installation and dependencies (e.g. Alex Birsan)
Environments

• Both pip and conda support environments
  - venv
  - conda env

• Idea is that you can create different environments for different work
  - environment for cs503
  - environment for research
  - environment for each project
Assignment 5

• Scripts and Modules
• Write a three modules in a Python package with methods to process Pokémon data
• Write a script to retrieve Pokémon information via command-line arguments
• MaxCP formula fixed by 2021-02-28
• Turn in a zip file with package
• No notebook required, but useful to test your code as you work
  - %autoreload or importlib.reload
What is the difference between an array and a list (or a tuple)?
Arrays

• Usually a fixed size—lists are meant to change size
• Are mutable—tuples are not
• Store only one type of data—lists and tuples can store anything
• Are faster to access and manipulate than lists or tuples
• Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.
Why NumPy?

- Fast **vectorized** array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with *if*- *elif*- *else* branches
- Group-wise data manipulations (aggregation, transformation, function application).

[W. McKinney, Python for Data Analysis]

D. Koop, CSCI 503, Spring 2021
import numpy as np
Creating arrays

- data1 = [6, 7, 8, 0, 1]
  arr1 = np.array(data1)
- data2 = [[1.5, 2, 3, 4], [5, 6, 7, 8]]
  arr2 = np.array(data2)
- data3 = np.array([6, "abc", 3.57]) # !!! check !!!
- Can check the type of an array in dtype property
- Types:
  - arr1.dtype # dtype('int64')
  - arr3.dtype # dtype('<U21'), unicode plus # chars
Types

• "But I thought Python wasn't stingy about types…"
• numpy aims for speed
• Able to do array arithmetic
• int16, int32, int64, float32, float64, bool, object
• Can specify type explicitly
  - arr1_float = np.array(data1, dtype='float64')
• astype method allows you to convert between different types of arrays:
  arr = np.array([1, 2, 3, 4, 5])
  arr.dtype
  float_arr = arr.astype(np.float64)
numpy data types (dtypes)

<table>
<thead>
<tr>
<th>Type</th>
<th>Type code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8, uint8</td>
<td>i1, u1</td>
<td>Signed and unsigned 8-bit (1 byte) integer types</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>i2, u2</td>
<td>Signed and unsigned 16-bit integer types</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>i4, u4</td>
<td>Signed and unsigned 32-bit integer types</td>
</tr>
<tr>
<td>int64, uint64</td>
<td>i8, u8</td>
<td>Signed and unsigned 64-bit integer types</td>
</tr>
<tr>
<td>float16</td>
<td>f2</td>
<td>Half-precision floating point</td>
</tr>
<tr>
<td>float32</td>
<td>f4 or f</td>
<td>Standard single-precision floating point; compatible with C float</td>
</tr>
<tr>
<td>float64</td>
<td>f8 or d</td>
<td>Standard double-precision floating point; compatible with C double and Python float object</td>
</tr>
<tr>
<td>float128</td>
<td>f16 or g</td>
<td>Extended-precision floating point</td>
</tr>
<tr>
<td>complex64,</td>
<td>c8, c16,</td>
<td>Complex numbers represented by two 32, 64, or 128 floats, respectively</td>
</tr>
<tr>
<td>complex128,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex256</td>
<td>c32</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>?</td>
<td>Boolean type storing True and False values</td>
</tr>
<tr>
<td>object</td>
<td>1</td>
<td>Python object type; a value can be any Python object</td>
</tr>
<tr>
<td>string_</td>
<td>S</td>
<td>Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'</td>
</tr>
<tr>
<td>unicode_</td>
<td>U</td>
<td>Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')</td>
</tr>
</tbody>
</table>
Array Shape

- Our normal way of checking the size of a collection is... `len`
- How does this work for arrays?
  - `arr1 = np.array([1,2,3,6,9])`
    ```python
    len(arr1)  # 5
    ```
  - `arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])`
    ```python
    len(arr2)  # 2
    ```
- All dimension lengths → shape: `arr2.shape` # (2,4)
- Number of dimensions: `arr2.ndim` # 2
- Can also reshape an array:
  - `arr2.reshape(4,2)`
  - `arr2.reshape(-1,2)` # what happens here?
Speed Benefits

• Compare random number generation in pure Python versus numpy

• Python:
  - import random
    %timeit rolls_list = [random.randrange(1,7) for i in range(0, 60_000)]

• With NumPy:
  - %timeit rolls_array = np.random.randint(1, 7, 60_000)

• Significant speedup (80x+)
Array Programming

• Lists:
  
  ```python
  c = []
  for i in range(len(a)):
      c.append(a[i] + b[i])
  ```

• How to improve this?
Array Programming

- Lists:
  - `c = []`
    - `for i in range(len(a)):`
      - `c.append(a[i] + b[i])`
  - `c = [aa + bb for aa, bb in zip(a, b)]`

- NumPy arrays:
  - `c = a + b`

- More functional-style than imperative
- **Internal iteration** instead of external
Operations

• \( a = \text{np.array}([1,2,3]) \)
  \( b = \text{np.array}([6,4,3]) \)

• (Array, Array) Operations (Element-wise)
  - Addition, Subtraction, Multiplication
  - \( a + b \) # array\([7, 6, 6]\)

• (Scalar, Array) Operations (Broadcasting):
  - Addition, Subtraction, Multiplication, Division, Exponentiation
  - \( a ** 2 \) # array\([1, 4, 9]\)
  - \( b + 3 \) # array\([9, 7, 6]\)
More on Array Creation

- **Zeros:** `np.zeros(10)`
- **Ones:** `np.ones((4, 5))` # shape
- **Empty:** `np.empty((2, 2))`
- `_like versions:` pass an existing array and matches shape with specified contents
- **Range:** `np.arange(15)` # constructs an array, not iterator!
Indexing

• Same as with lists plus shorthand for 2D+
  - arr1 = np.array([6, 7, 8, 0, 1])
  - arr1[1]
  - arr1[-1]

• What about two dimensions?
  - arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])
  - arr[1][1]
  - arr[1,1] # shorthand
In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array $\text{arr3d}$:

$$\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
10 & 11 & 12
\end{bmatrix}$$

$\text{arr3d}[0]$ is a $2 \times 3$ array:

$$\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6
\end{bmatrix}$$

Both scalar values and arrays can be assigned to $\text{arr3d}[0]$:

```python
old_values = arr3d[0].copy()
arr3d[0] = 42
arr3d[0] = old_values
```

[W. McKinney, Python for Data Analysis]
Slicing

• 1D: Similar to lists
  - arr1 = np.array([6, 7, 8, 0, 1])
  - arr1[2:5] # np.array([8,0,1]), sort of

• Can **mutate** original array:
  - arr1[2:5] = 3 # supports assignment

• Slicing returns **views** (copy the array if original array shouldn't change)
  - arr1 #
  - arr1.copy() or arr1[2:5].copy() will copy
Slicing

• 2D+: comma separated indices as shorthand:
  - arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])
  - a[1:3, 1:3]
  - a[1:3, :] # works like in single-dimensional lists

• Can combine index and slice in different dimensions
  - a[1, :] # gives a row
  - a[:, 1] # gives a column
How to obtain the blue slice from array \texttt{arr}?
How to obtain the blue slice from array \( arr \)?

\[ arr[:2,1:] \]