Using Python for Data

Useful Packages

- `astropy` : Includes functions for reading/writing data files (including `.fits`), cosmology calculations, astronomical constants and coordinate systems, image processing, and much more
- `numpy` : Adds ability to deal with multi-dimensional arrays and vectorized math functions
- `scipy` : Extends `numpy` by adding common scientific functions such as ODE integration, statistical analysis, linear algebra, and FFT
- `matplotlib` : A useful plotting package
- `pandas` : Package for dealing with data tables
- `astroML` : Common statistical analysis and machine learning tools used in astronomy
- `scikit-learn` : More machine learning tools written in python

Installing python

The easiest way to install python on any OS is to use anaconda python. This will install a local version of python on your system so you don't need to worry about needing admin to install new packages. Most of the packages listed above are installed by default with anaconda. For this class we will be using python 3, and I recommend you use this version for you research (unless you have a very good reason to use python 2).

Note

As of October 2019 python 2.7 is officially depreciated and will only receive security updates and in 2023 python 3.7 was officially depreciated as well. Many of the major packages listed above have already dropped python 2 support are are starting to drop support of python 3.7 and lower.

Text editors

Although there are numerous IDEs (e.g. IDLE, Spyder) for python, for most everyday use you will likely be writing python code in a text editor and running your programs via the command line. In this case it is important to have a good text editor that supports syntax highlighting, live linting (syntax and style checking), and is easy to configure the way you want. I can highly recommend VScode as a free text editor with all the features above.

For python coding in VScode you will want to install the `Python` extension by Microsoft (you will be prompted to install it when you first open a .py file) and the `Jupyter` extension by Microsoft. Other useful extensions are the `Excel Viewer` extension for easier viewing CSV files, `open in browser` for and option to open HTML files in your browser, `MyST-Markdown` for rendering markdown files, and `Code Spell Checker` for basic spell checking.

Coding style

What is a coding style? Beyond the syntax of a coding language, a coding style is a set of conventions that can be followed to make it easier for other developers (including your future self) to read you code and to understand the intention behind your code. For python coding the style most developers use has it basis in PEP 8.

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.
>
> However, know when to be inconsistent – sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Here are some examples of PEP 8 conventions:

- Use 4 spaces to indent lines (rather than a tab)
- A max line limit of 79 characters (preferred by people who use command line editors, I typically override this to be higher)
- Constants are defined at the module level with names in `ALL_CAPS`
- Class names should normally use the `CapWords` convention
- Function names should be `lowercase`, with words separated by underscores as necessary to improve readability

Basic syntax examples

For a general overview of python's syntax head over to codecademy and take their interactive tutorial. In this class we will only be covering what is necessary for data analysis.

importing packages

Any package or code from another `.py` file can be imported with a simple `import` statement. By default all imported code has its own name space, so you don't have to worry about overwriting existing functions. The final line of this code block is a "magic" `Jupyter` function needed to make interactive plots inside of `Jupyter notebooks`.

In [ ]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
%matplotlib inline
```

data containers

Data inside of python can be stored in several different types of containers. The most basic ones are:

- `list`: an indexed data structure that can hold any objects as an element
- `tuple`: same as a `list` except the data is immutable
- `dictionary`: objects stored as a `{key: value}` set (note: any immutable object can be used as a key including a tuple)

In [2]:
```
example_list = [1, 2, 3]
example_tuple = (1, 2, 3)
example_dict = {'key1': 1, 'key2': 2, ('key', 3): 3}
```

Elements in these objects can be accessed using an zero-based index (`list` and `tuple`) or key (`dict`).

In [3]:
```
print(example_list[0], example_list[-1])
print(example_tuple[1])
print(example_dict['key1'], example_dict[('key', 3)])
```

```
1 3
2
1 3
```

Each of these objects have various methods that can be called on them to do various things. To learn what methods can be called you can look at the python documentation (e.g. https://docs.python.org/3/tutorial/datastructures.html) or you can inspect the object directly and use python's `help` function to get the doc string.

Note: Methods that start with `__` or `_` are private methods that are not designed to be called directly on the object.

In [ ]:
```
print(dir(example_list))
print('\n\n')
help(example_list.pop)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__deli
tem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_sub
class__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__'
, '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'co
py', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']



Help on built-in function pop:

pop(index=-1, /) method of builtins.list instance
    Remove and return item at index (default last).

    Raises IndexError if list is empty or index is out of range.
```

Slicing lists

Many times it is useful to slice and manipulate lists:

In [ ]:

```python
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
# print the first 3 elements
print(a[:3])
# print the middle 4 elements
print(a[3:7])
# print the last 3 elements
print(a[7:])
# you can also use neg index
print(a[-3:])
# print only even index
print(a[::2])
# print only odd index
print(a[1::2])
# print the reverse list
print(a[::-1])
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2]
[3, 4, 5, 6]
[7, 8, 9]
[7, 8, 9]
[0, 2, 4, 6, 8]
[1, 3, 5, 7, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Looping over `list` s and `dict` s

There are several ways to loop over a `list` or `dict` depending on what values you want access to.

In [6]:

```python
# loop over values in a list
for i in example_list:
    print(i)
print('=========')

# loop over values in a list with index
for idx, i in enumerate(example_list):
    # print('{0}: {1}'.format(idx, i))
    print(f'{idx}: {i}')
print('=========')

# loop over keys in dict
for i in example_dict:
    print(i)
print('=========')

# loop over values in dict
for i in example_dict.values():
    print(i)
print('=========')

# loop over keys and values in dict
for key, value in example_dict.items():
    # print('{0}: {1}'.format(key, value))
    print(f'{key}: {value}')
```