Computational Finance

Week 2: Dealing with Data

Outline

- Broadcasting
- Matrix multiplication
- Debugging
    - Methods
    - In Jupyter

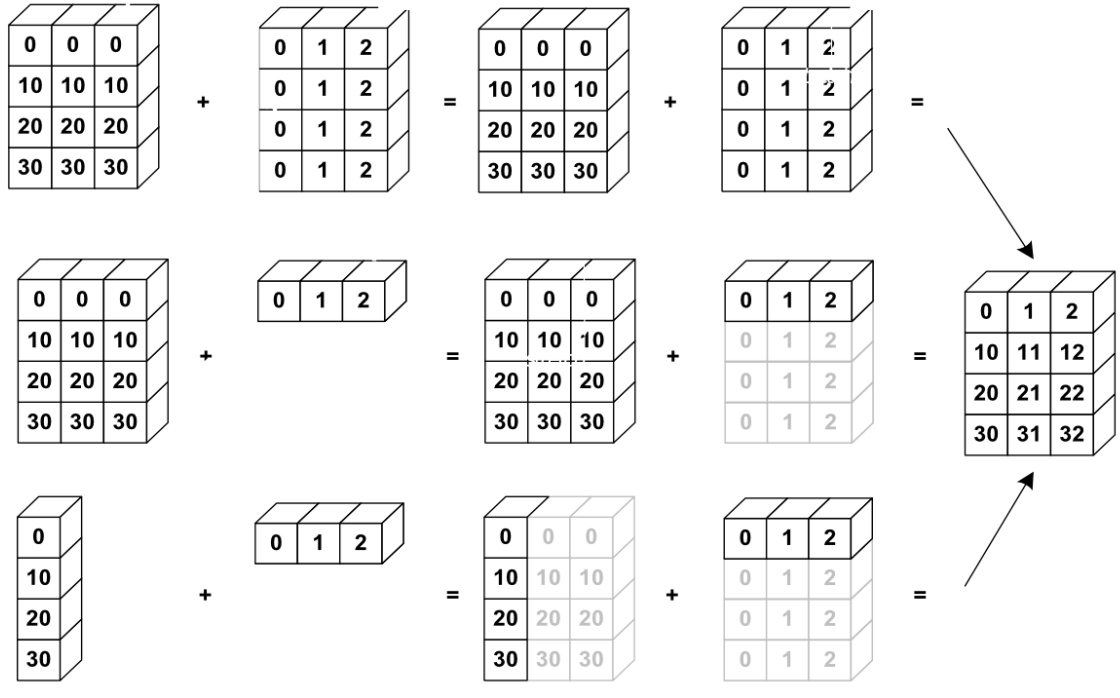
Broadcasting

- Broadcasting is a useful functionality of numpy, but can be tricky to understand.
- "The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations".
- "Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes"
- See NumPy documentation (https://numpy.org/doc/stable/user/basics.broadcasting.html) for more details

- NumPy compares the shapes of two arrays dimension-wise.
- It starts with the trailing (i.e. rightmost) dimensions, and then works its way left.
- Two dimensions are compatible if
    - they are equal, or
    - one of them is 1 (or not present).
- *Tip*: write down the dimensions and draw the arrays.

In [ ]:
```
import numpy as np
```

- The idea of broadcasting in a picture:



What is the shape?

In [ ]:
```
a = np.arange(8).reshape(2,4)  # (2,4)
b = np.arange(4)               # (4,)
a
```

In [ ]:
```
b
```

In [ ]:
```
a + b
```

What is the shape?

In [ ]:
```
a = np.arange(8).reshape(4,2)  # (4,2)
b = np.arange(4).reshape(1,4)  # (1,4)
a,b;
```

In [ ]:
```
a + b
```

What is the shape?

In [ ]:
```
a = np.arange(8).reshape(4,2)  # (4,2)
b = np.arange(4).reshape(4,1)  # (4,1)
a, b;
```

In [ ]:
```
a + b
```

What cell does NOT throw an error?

In [ ]:
```
a = np.arange(8).reshape(4,2)   # (4,2)
b = np.arange(4)                # (4,)
```

In [ ]:
```
a
```

In [ ]:
```
b
```

In [ ]:
```
a + b
```

In [ ]:
```
a + b[:, np.newaxis]
```

In [ ]:
```
a + b[np.newaxis, :]
```

Dimensions of b

In [ ]:
```
b   # (4,)
```

In [ ]:
```
b[:, np.newaxis]  # (4,1)
```

In [ ]:
```
b[np.newaxis, :]  # (1,4)
```

In [ ]:
```
np.newaxis == None
```

In [ ]:
```
b[None, :]  # (1,4)
```

Matrix multiplication

- Note than NumPy has reserved * for element-by-element multiplication.
- We cannot use * for matrix multiplication (https://en.wikipedia.org/wiki/Matrix_multiplication#Definition)
- Instead, one can use the NumPy-function `dot` or `@` in Python 3.x:

In [ ]:
```
A = np.arange(6).reshape(2,3) #(2,3)
b = np.arange(3)              #(3,)
A, b
```

In [ ]:
```
A*b
```

In [ ]:
```
A@b
```

In [ ]:
```
A@b[:,np.newaxis]
```

The module `numpy.linalg` has a standard set of matrix decompositions and functions calculating things like inverse (`inv`), trace (`trace`), determinant (`det`) and eigenvalues and eigenvectors (`eig`):

In [ ]:
```
A = np.arange(4).reshape(2,2) #(2,2)
A
```

In [ ]:
```
np.linalg.inv(A)
```

Finally, in order to transpose a matix/vector we could use:

In [ ]:
```
A.T
```

Debugging

- 'Definition': find and fix mistakes (a.k.a. bugs) in code
- Bad news
    - Inevitable and frustrating
    - Hard to predict how long it takes to solve
- Good news
    - Tools and methods available
    - Gain experience and learn from them

Debugging methods

Credit: mostly based on Christoph Deil's talk at PyConDE (https://github.com/cdeil/pyconde2019-debugging).

Read the code

- Can you spot (obvious) mistakes/typos?
- Need to know syntax and structure, e.g. how to define a function.
- Check documentation of Jupyter / Python / package
- Use `help()` or `Shift+Tab`

Read traceback

- Errors are your friend. Mistakes without error are harder to spot and may be overlooked.
- Error message holds useful information
    - *what* type of error.
    - *where* was it raised, at what line of code.
- Bug is usually in your code, not in the Python packages such as numpy or pandas.

In [ ]:

```
a = np.arange(8).reshape(4,2)   # (4,2)
b = np.arange(4)                # (4,)
a+b
```

Print intermediate variables

- Seems easy, but ...
- Clutters code and output
- Need to choose what to print and where

Debugger

- Execute code line by line
- Breakpoints
- Visual debugger
    - Available in IDEs such as PyCharm (https://www.jetbrains.com/pycharm/) and Visual Studio Code (https://code.visualstudio.com/)
- Command line debugger

Rubber duck debugging

- Explain your problem to a *rubber duck*.
- Often then you realize what the problem is.
- Doesn't have to be a duck per se:

Debugging in Jupyter

- Visual debugger under development, see GitHub (https://github.com/jupyterlab/debugger).
- Command line debugger: `pdb` , python debugger (docs (https://docs.python.org/3/library/pdb.html), debugger commands (https://docs.python.org/3/library/pdb.html#debugger-commands)).
- Post mortem analysis using `%debug` (docs (https://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-debug)).
    - Step into function *after* the error occurred.

In [ ]:

```
a = np.arange(11)
b = np.arange(5,15)
foo = a > 5
b[foo]  #feeding a boolean index for slicing
```

In [ ]:
```
%debug
```

- Breakpoints stop/pause the code so you can inspect the state
    - Set breakpoints using `set_trace()`
    - Alternative: `breakpoint()` from Python 3.7

In [ ]:

```
from IPython.core.debugger import set_trace   # import set_trace()
a = np.arange(11)
b = np.arange(5,15)
set_trace()  # breakpoint
foo2 = a > 5
b[foo2]  #feeding a boolean index for slicing
```

In [ ]:

```
mysum = 0
for i in range(5):
    set_trace()
    mysum += i
```

So... what should we do?

- First design, then write code
- Write simple clean code
- Add comments and docstrings
- Test your code at intermediate steps (functions, loops)

When you run into a bug

- Read code and error message
- Check for common mistakes
    - Typos (may be hard to spot!)
    - Syntax errors
    - Wrong shape
    - Wrong type
    - Overwriting variables
    - Not assigning output
    - Loop variable
    - ...

- Inspect variables at intermediate steps (functions, loops)
- Pen and paper
- Clean slate

- More tips: Debugging for beginners (https://docs.microsoft.com/en-us/visualstudio/debugger/debugging-absolute-beginners?view=vs-2019) by Microsoft, aimed at VS Code users, but contains general tips on steps when debugging.

In [ ]: