Programming in Python

The course will require you to use the Python programming language, and will heavily use array-based computation using the NumPy library.

Why Python: Python is widely-used, general-purpose programming language that provides access to a large number of datascience and machine learning frameworks. Unlike some special-purpose statistical languages, you'll have to learn to routinely import some modules, as outlined below. But then you'll find that the language does everything you need, and is widely supported.

What about other languages? Fashions can change quickly. Until 2016, Lua was used with Torch as the main machine learning framework at Facebook AI and Google DeepMind, and so might have seemed like a safe and good option. However, there is now next to no community around this framework. A lot of machine learning code used to be written in Matlab, or its free-software equivalent Octave. These languages are quicker to get started with than Python and NumPy, but are less good for writing larger programs, and don't have the same support by the top machine learning frameworks. The R language also has its place for statistical work. Newcomers usually find it quirky, however it has a large collection of well-documented statistical packages in CRAN, and is a good choice if you primarily want to use existing statistical toolboxes. If you want to write compiled code, you might look at using the C++ library Eigen (as used internally by TensorFlow).

The important thing is to learn the principles of array-based computation for machine learning. If you start with Python and NumPy, you should be able to rapidly generalize to whatever tool you need to use in the future.

1 Getting started with Python

Python and its associated scientific libraries are installed on the Informatics DICE system.

If installing on your own machine, we recommend trying the Anaconda distribution, unless the package manager you normally use to install software has well-maintained Python packages. Some software distributions come with fairly old Python packages, whereas Anaconda usually "just works". Whatever route you take, you'll want at least Python, NumPy, SciPy, and Matplotlib. You should install Python 3 rather than Python 2 (more below).

If you don't already know the basics of Python, you should first find a Python tutorial at your level, and work through it. The official Python tutorial is a good start. (You don't need the more advanced topics, like classes, or to work through all of the standard library examples.) Then you would need to learn the NumPy and Matplotlib libraries. Again, there are many tutorials online. You might start with the official quickstart guide. For more, you could work through some of scipy-lectures.org, which aims to be "One document to learn numerics, science, and data with Python".

You can use Python interactively from the **ipython command-line program**. From there you can type `%paste` to run code in the clipboard, or use the `%run` command to run code stored in a file. If you get an error, you can use `%debug` to enter a debugger. If you start ipython with `ipython3 --matplotlib` then plotting works smoothly: there's no need for `plt.show()` commands, and plot windows don't cause the interpreter to hang. Alternatively type `%matplotlib` after starting ipython.

Those that like a graphical environment could try Spyder. There are also popular heavy-weight commercial environments such as PyCharm.

IPython or Jupyter notebooks are becoming popular, and are used in some other courses. If you like the notebook interface, feel free to use it yourself. They're great for producing a demonstration of how to use a library, or for working notes where you can save results inline.

However, they aren't a good way of holding the main code for a project, or for collaboration. Notebooks save results and code in one file, which doesn't work well with version control, and if you send someone a notebook, you're forcing them to launch a server and open a web-browser, rather than using the development environment of their choice. Make sure you are also able to work with code stored in `.py` files.

1.1 Commonly-used Python modules

If you use Python, you will use NumPy extensively. The standard way to use this module is

```
import numpy as np
```

Then some example code would be:

```
A = np.random.randn(3, 3)
matrix_product = np.dot(A, A) # simply "A @ A" with python >=3.5
```

Python examples might not always specify the `import` line, but you'll need it if the code refers to `np.something`. Similarly if an example uses `plt`, a Matlab-like plotting interface, you'll need to import it as follows:

```
import matplotlib.pyplot as plt
```

Some people reduce the amount of typing they need to do with:

```
from numpy import *
from numpy.random import *
from matplotlib.pyplot import *
```

which means code can directly call functions like `dot()` and `plot()` without a "`np.`" or "`plt.`" prefix. Sometimes short Matlab snippets work unaltered in Python this way (although care is required). Ready access to the functions is convenient for interactive use, but importing a large set of functions is usually considered poor practice in "real code". For example Python's `sum()` and `max()` and NumPy's `np.sum()` and `np.max()` could become confused with each other, which can lead to subtle bugs.

1.2 Python/NumPy Arrays, matrices, vectors, lists, tuples, ...

One reason that numerical computation with Python is more complicated for beginners than dedicated numerical languages like Matlab is the larger number of types you have to deal with immediately.

Python's usual `tuple` and `list` types don't provide convenient array-based arithmetic operations. For example

```
xx = [1, 2, 3] # python list
print(xx*3)           # prints [1, 2, 3, 1, 2, 3, 1, 2, 3]
print((1,2) + (3,4))  # prints (1, 2, 3, 4)
```

You will use the list or tuple types to initialize NumPy arrays, and also as containers of NumPy arrays of different shapes.

NumPy has a "matrix" type (created with `np.matrix`), which we strongly recommend you avoid completely (as does the wider NumPy community). Standard practice is to use NumPy *arrays* for all vectors, matrices, and larger arrays of numbers. Attempting to mix NumPy matrix and array types in your code is likely to lead to confusion and bugs.

One way to ensure you're dealing with NumPy arrays is to convert to them at the top of functions you write:

```
def my_function(A):
    A = np.array(A) # does nothing if A was already a numpy array
    N, D = A.shape # now works, even if A was originally a list of lists
```

Unlike Matlab, NumPy distinguishes between scalars, vectors, and matrices. If you're going to use NumPy, you should know (or work out) what the following code outputs, and why:

```
A = np.random.randn(3, 2)
print(A.shape)
print(np.sum(A,1).shape)
print(np.sum(A).shape)
```

If some NumPy code expects an array of shape `(N,)`, a vector of length `N`, it might not work if you give it an array of shape `(N,1)` or `(1,N)` (and vice-versa). You can convert between vectors and 2D arrays using `np.reshape`, `np.ravel()`, and indexing tricks.

1.3 Broadcasting

A common NumPy task is to subtract a vector `rv` from every row of a matrix `A` stored in an array:

```
# For shape (N,M) array A, and shape (M,) array rv
A - rv   # or more explicitly: A - rv[None,:]
```

To subtract a vector `cv` from every column:

```
# for shape (N,) array cv
A - cv[:,None]
```

Here "`None`" creates a new axis: `cv[:,None]` is a 2-dimensional array with shape `N,1`. The single column is automatically "broadcast" by NumPy across the M columns of `A`. If you didn't expand `cv` into a 2-dimensional array, the subtraction would fail.

You can use `newaxis` from the numpy module instead of `None`, which is more explicit. However, I don't always want to have to import `newaxis`, and `np.newaxis` is too long to repeat many times in code that does a lot of indexing. NumPy isn't going to break the use of `None`, because lots of code uses it and it's documented.

1.4 "Assignment" and pass-by-reference

This section is about a common misunderstanding that can lead to incorrect Python code.

In Python "=" is used for "assignment", but when there's just a variable name on the left, a more precise description is that it's for "attaching the name on the left-hand side to the object on the right-hand side". A simple example is:

```
A = np.ones((2, 2))
B = A
B[0, 0] = 25
print(A)
```

The second line "`B = A`" attaches the name `B` to the same object that the name `A` is already attached to — a 2×2 array of ones.

"`B[0, 0] = 25`" modifies the first element of the underlying object, so both `A` and `B` are changed.

If you don't want to accidentally change arrays, write "`B = A.copy()`" not "`B = A`". Also for *slices*: "`first_row = A[0].copy()`"

For objects other than NumPy arrays, you might need:

```
import copy
B = copy.deepcopy(A)
```

Similarly, arguments to functions are references to objects that might have other names. You shouldn't alter the original objects, unless the caller definitely knows that the arguments could be modified. Here's one pattern to make a function safer to use:

```
def my_function(A, in_place=False):
    if not in_place:
        A = A.copy()
    A += 1 # ... modify A
    return A
```

```
A = np.ones((2, 2))
B = my_function(A)                # A and B are different
B = my_function(A, in_place=True) # A and B are the same (saves memory)
```

Or you could just always take the copy, losing the ability to save some memory and time, but making the code simpler.

If you don't take copies, you'll have to be pretty careful to track when names share objects. For example, it takes a moment to be sure what the following does:

```
A = np.ones((2, 2))
B = A
B = B + 6
print(A)
B += 6
print(A)
```

The right-hand side of "`B = B + 6`" creates a new object[1]. So this line *doesn't* affect the object referred to by `A`. Moreover, this line attaches `B` to a different object than `A`. Therefore, the line "`B += 6`" doesn't affect `A` either — although it would have done without the "`B = B + 6`" line!

So "`B = A.copy()`" isn't necessary in this example. But it would have been a good idea for clarity, and could avoid bugs later when the code is altered.

If none of that is confusing: congratulations! You're probably an experienced programmer.

If it is confusing, you're not alone. Invest time to work through examples like those above at a Python prompt. Also try out each part of code you write on small example arrays to check it does what you think it does. (Which experts should do too!)

1.5 Python 2 vs Python 3

The migration from Python 2 to 3 has been slow and painful. As recently as 2016 OpenAI stated many researchers were still using Python 2.7. However, Python 2 is no longer supported, so you should definitely use Python 3 for new code.

The main change in Python 3 is Unicode string handling, which isn't relevant for the sort of code we'll write in this course. The minor issue you'll have to deal with in practice is avoiding Python 2 print statements:

```
print "Hello World!"  # Python 2 code that will crash in Python 3
```

Add parenthesis around the string as follows:

```
print("Hello World!")  # Works in both Python 2 and Python 3
```

Replace any more complicated Python 2 print statements with Python 3 style print functions. You might see `from __future__ import` lines at the top of code, which are to keep these examples working in Python 2 as well. For example:

```
from __future__ import print_function  # not needed any more

print('thing1', 'thing2', sep=', ')
```

Python 3.5 came with a matrix multiply operator `@` which performs `np.matmul`. You can often write `A @ B` instead of `np.dot(A, B)`. However, be careful: `np.matmul` has different broadcasting rules and doesn't work with scalars. There is also no easy way to get the `@` operator in earlier versions of Python, so examples in the notes tend to use `np.dot` to ensure broad compatibility. But if you're using Python $\geq$ 3.5, you could go ahead and try out the `@` operator in your own code.

1. *In theory*, if `B` referred to an object from a different library than NumPy, "`+`" *could* modify `A` in place and return the original object. In that case, `A` would be modified, and would still refer to the same object as `B`. Ouch! Fortunately, the classes in most libraries are written so that "`+`" doesn't have surprising side-effects.