Module 7: Numpy and Plotting

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-07-start.ipynb
```

Discuss the previous module with your neighbour.

- How can we read and write files?
- How many different ways can we read a CSV file?

NumPy is a tool that is very useful in high-performance scientific computing.

It will be really important in PHYS 249, where it is used for to do **linear algebra**, in particular arithmetic using **vectors** and **matrices**. It doesn't let us do anything we couldn't do without it, but it makes certain things considerably easier.

NumPy is a module, so we need to import it. But since we're going to use it a lot, we're going to "rename" it as we import it. We write:

```
import numpy as np
```

Now `np` refers to the `NumPy` module.

NumPy Arrays

In plain Python, we work mostly with **lists**.

In NumPy, we instead work with **arrays**. We will use the type `np.ndarray`. We can create one from a `list[float]` using the `np.array` function:

```
np.array([3.14, 2.717, 1.414, 42.0]) ⇒ array([ 3.14 ,  2.717,  1.414, 42.   ])
```

Recall: a list can contain values of one type, a mix of types, or more.

```
mylist = [2, "four", 3.14]
```

A `np.ndarray` **can** contain a mix of values:

```
np.array(mylist) ⇒ array(['2', 'four', '6.0'], dtype='<U32')
```

but for performance reasons we want it to be all values of one type: `float` or `int`.

(Technically, `numpy.float64` or `numpy.int64`, but we ignore the distinction.)

Working with NumPy Arrays

The big difference with `np.ndarray` values is that everything is **componentwise**, or **item-by-item**:

```
np.array([1,2,3]) + np.array([20,15,10])
⇒ np.array([1+20, 2+15, 3+10])
⇒ np.array([21, 17, 13])
np.array([1,3,5,1]) * np.array([7,6,2,2])
⇒ np.array([1*7, 3*6, 5*2, 1*2])
⇒ np.array([7, 18, 10, 2])
np.array([1, 3, 5, 1]) * 4
⇒ np.array([1*4, 3*4, 5*4, 1*4])
⇒ np.array([4, 12, 20, 4])
np.array([2, 6, 6]) < np.array([3, 5, 7])
⇒ np.array([2 < 3, 6 < 5, 6 < 7])
⇒ np.array([True, False, True])
```

For our purposes we will only work with arrays that have the same length, or with an array and a single number.

```
import numpy as np
def sp1(x: np.ndarray) -> np.ndarray:
    """Return the square of each
    item in x, plus one."""
    return x**2 + 1.0

check.expect("SP1",
             sp1(np.array([2., 3., 1.])),
             np.array([5., 10., 2.0]))
```

Exercise

Using this, write a one-line function `double` that takes a `np.ndarray` and returns a `np.ndarray` with each value doubled.

It can be useful to have evenly spaced values within some interval.

E.g to have 6 evenly spaced values between 7 and 8, like `[7.0, 7.2, 7.4, 7.6, 7.8, 8.0]`.

You might try to do this using `range`, but it doesn't work. `range(7.0, 8.2, 0.2)` raises an error.

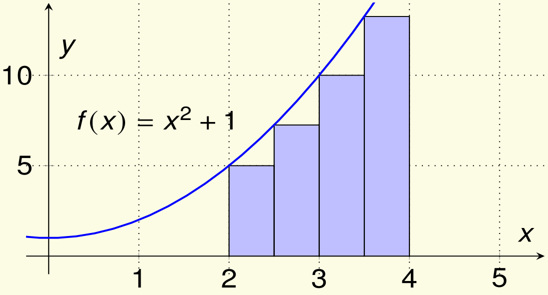
```
## Uncomment this and try it; we get an error.
# range(7.0, 8.2, 0.2)
```

The function `np.linspace` does what we want: `linspace(start, stop, count)` will return a `np.ndarray` containing `count` values between `start` and `stop`.
`np.linspace(7,8,6)` ⇒ `array([7. , 7.2, 7.4, 7.6, 7.8, 8. ])`

By default this **includes** the endpoint. Turn it off with the flag `endpoint=False`:
`np.linspace(7,8,5, endpoint=False)` ⇒ `array([7. , 7.2, 7.4, 7.6, 7.8])`

Example: Approximating the area under a curve

The area of a rectangle is $b \times h$ where b and h are the base and height.
We can estimate the area of any weird shape by adding up a lot of little rectangles.
The area of $f(x) = x^2 + 1$, using 4 bins between 2.0 and 4.0, is approximately:



$$5 \times 0.5 + 7.25 \times 0.5 + 10 \times 0.5 + 13.25 \times 0.5 = 17.75$$

Exercise

Write a function `approx_area(f: callable, x0: float, x1: float, nbins: int) -> float`.
The function returns an approximation of the area of between `f` and the x-axis, between `x0` and `x1`, using `nbins` bins. **Use no loops!**

1. Find an expression to give all x values. **Use `print` to verify that it is right.**
2. From that, get all the heights. **Use `print` to verify that it is right.**
3. Multiply by the width of a bin to get all the areas, then add them up with `sum`.

Vectorizing

This looks all good. But try this:

```
approx_area(math.sin, 0.0, math.pi, 1000)
```

It "should" return something close to `2.0`. But instead:

```
TypeError: only size-1 arrays can be converted to Python scalars
```

`math.sin` wants a **`float`**, and is unhappy with this `np.ndarray`.

The NumPy library has its own "vectorized" versions of these functions that work on arrays:

`approx_area(np.sin, 0.0, np.pi, 1000)` ⇒ `1.9999983550656624`

```python
def box(n: float) -> float:
    """Return 1 for n between 2 and 5, and 0 otherwise."""
    if 2 < n < 5:
        return 1
    else:
        return 0
```

box(4) ⇒ 1

box(np.array([1,2,3,4,5,6]))
→ "ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()"

Since this function doesn't do only vector stuff, it's unhappy, just like `math.sin` was unhappy.

To make such a function, you can use `np.vectorize`. It takes a **`callable`**, and returns a new **`callable`** that works nicely with a `np.ndarray`:

```
vbox = np.vectorize(box)
vbox(np.array([1,2,3,4,5,6])) ⇒ np.array([0,0,1,1,0,0])
```

Exercise

From the start code, run the following code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/pyplot.ipynb
```

Don't attempt to type this in by hand. Use the module start code!

Now you have the pyplot tutorial; look in `pyplot.ipynb`.

Some key things to see:

- Use `plt.plot` with a pair of equal-sized iterables (lists or arrays) to plot lines or points.
- Use `plt.scatter(x, y, s=sizes)` to plot points with controllable sizes.
- Multiple plots on the same figure by calling `plt` functions repeatedly.
- To start a new blank plot, call `plt.figure()`.
- There are lots of fancy options for style. Use them freely; I won't test you on them.

Plotting and Data

From the pyplot tutorial we now know how to make certain simple plots. Using `plt.plot` with two lists (or arrays) of the same length, we can get nice plots.

Usually we want to plot **data**. So let's get some data from a file.

Look at the file `w-plot.csv`

Exercise

1. Use the `csv` module to load the data from this file, in such a way that you have two `list[int]`. I suggest you call your variables `x` and `y`.

For example this:	should print:
`print(x)`	`[6, 1, 5, 2, ...]`
`print(y)`	`[11, 15, 9, 13, ...]`

2. Use `plt.plot` to plot these values.

Hint

If you instead see `['6', '1', '5', '2', ...]`, it means you read the strings from the file, but never converted to integers.
Always remember to use `int` or `float` when reading numeric data from CSV.

Once we have a collection of **measurements**, we want to propose a **model** that describes the measurements. Often a model will have some **constants** or **parameters** that we need to determine.

For example, we might suspect a linear relationship between the **mass** of an object and the **force of gravity** on it. But the slope of that line depends on other things.

We will look at a new file to learn about how to fit curves to data.

Exercise

To get the file run the command:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/curve-fitting.ipynb
## Look through this notebook to see how to fix a polynomial to data.
## Keep this notebook to refer to later.
```

Here we use `numpy.polynomial.Polynomial.fit` to fit a polynomial to some data.

This tells us the parameters of our model.

Exercise

In Jupyter, download the following file, and load the data:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/some-data.csv
## Use the csv module to load the data in this file, and plot it.
## Fit polynomials of different degree to it.
```

Plot the data.

What shape is it? Try fitting polynomials of different degree to this dataset.

Practice Curve Fitting

You can generate a `np.ndarray` containing random values on $[0, 1)$, using `np.random.rand`:

`np.random.rand(5)` $\Rightarrow$ `array([0.85082926, 0.222596 , 0.83213829, 0.60130446, 0.06038033])`

Exercise

Generate some random x values with `x = np.random.rand(10)`.
Then pick constants m and b to make matching y values for a line, using the expression $y = mx + b$.
Plot these with `'o'` to get points. Then fit a line through these points; you should get exactly the same m and b.

Exercise

Generate some random x values with `x = np.random.rand(10)`.
Then pick constants m and b to make matching y values for a line, using the expression $y = mx + b$.
Then add some zero-mean random noise to your y values by adding `np.random.rand(10) - 0.5` to it.
Plot these with `'o'` to get points. Then fit a line through these points; what do you get?

Module summary

- Use NumPy arrays to store collections of numbers.
- Do vectorized calculations on NumPy arrays.
- Draw plots using `plt.plot` and `plt.scatter`, based on functions or data.
- Use `numpy.polynomial.Polynomial.fit` to fit a polynomial to data.

Before we begin the next module:

- Read and complete the exercises in module 7 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 7 Review Quiz, due on Monday.