numpy_example

November 1, 2019

1 Using NumPy: Introduction and a worked example

Read through the following notes carefully, and run each of these cells to get an idea of how NumPy works.

We import `numpy` as below, and then we can access all Numpy functions from using `np.<function>`:

[ ]:
```
# np is the conventional name for numpy
import numpy as np
np.set_printoptions(suppress=True, precision=2) # configure printing
```

2 Basic array creation

`np.array()` takes a *sequence* and converts it into an `ndarray`; this can be, for example, an ordinary Python list:

[ ]:
```
## a simple vector: np.array converts iterable sequences to arrays
n_list = [1.0, 2.0, 3.0]
n_array = np.array(n_list)
print(n_array, type(n_array)) # this is now an array
```

2.1 Shape

`ndarrays` hold *rectangular* blocks of numbers. They can have any number of dimensions: 1D vectors, 2D matrices, 3D and higher tensors.

All ndarrays have a **shape**, which is the number of elements in each dimension. You can always find the shape of an existing array by calling `a.shape()` on an array `a`. This list is just three elements, so its shape is just `(3,)` (a 3 element vector):

[ ]:
```
print(n_array.shape)
```

2.1.1 Multidimensional arrays

This extends to creating multidimensional arrays, using nested sequences (e.g. list-of-lists). However, arrays must be **rectangular**; they cannot be *ragged*, where there are different numbers of columns per row.

[ ]:
```
# using nested lists to create a 2D array (a matrix)
a = np.array([
                [1.0, 2.0, 3.0],
                [0.0, 0.5, 0.0],
                [5.0, 0.0, 10.0]
               ])

print(a)
```

[ ]:
```
print(a.shape)
```

NumPy supports arbitrary dimensioned arrays. So we can create a `2x2x2` array of numbers:

[ ]:
```
a = np.array([
                [[1.0, 1.0],
                 [4.0, 4.0]],
                [[2.0, 2.0],
                 [3.0, 3.0]],
               ])
print(a)
```

[ ]:
```
print(a.shape)
```

2.2 Blank arrays

It's often useful to create arrays which are "blank", in the sense they are filled with equal values (particularly, filled with all zeros). There are array functions that will be produce such arrays very efficiently. They just take the shape of the desired array and return a fully populated array:

- `np.zeros(shape)` will create an array of a given **shape** and fill it with zeros.
- `np.ones(shape)` does the same and fills it with ones (1.0).
- `np.full(shape, value)` fills a new array with the given value.
- `np.empty(shape)` creates a new array, but doesn't fill it. This will have whatever was in the memory before, so while it's very efficient, it's only useful when you expect to overwrite the array values.

2.2.1 Shapes are tuples

Note: the `shape` is a **tuple**, written in round brackets like this: `(3,4)` or `(3,)`.

- **Don't** call `np.zeros(3,4)`: this calls `np.zeros` with two arguments
- **Do** call `np.zeros((3,4))`: this calls `np.zeros` with one argument, a shape tuple.

[ ]:
```
print(np.zeros((5,))) # a 5 element vector of zeros
```

[ ]:
```
print(np.zeros((3,3))) # a 3x3 matrix of zeros
```

[ ]:
```
print(np.zeros((6,2))) # a 6x2 matrix of zeros
```

[ ]:
```
print(np.ones((2,2))) #  a 2x2 matrix of ones
```

[ ]:
```
print(np.full((1,5,1), 9)) # a 1x5x1 array of 9s
```

[ ]:
```
np.full((2,2,2), -1.0) # 2x2x2 array filled with -1
```

[ ]:
```
print(np.empty(8,)) # 8 element vector -- note the values will be random junk!
```

2.3 Singleton axes

Note: a (1,5,1) array is different from a (1,5) array and different from a (5,) array. Array dimension sizes can be 1.

We call each dimension of an array an **axis**, and any **axis** which is just one element is called a **singleton axis**.

[ ]:
```
# these are all differently shaped arrays, even though they all contain exactly␣
 ↪five elements
print(np.zeros((1,5,1)).shape)
print(np.zeros((1,5)).shape)
print(np.zeros((5,)).shape)
```

2.3.1 zeros_like

It is often handy to create a new array which is the same type and shape as an existing array, but initialised to all zeros (or all ones). This is useful, for example, when accumulating counts in an array the same size as some original input (e.g. counting number of pixels that are blue or green)

`np.zeros_like()` does this. There are equivalent `ones_like` and `full_like` as well.

[ ]:
```
x = np.array([[1.0,2.0,3.0], [4.0,5.0,6.0]], dtype=np.float64)
print(x)
```

[ ]:
```
zeros_like_x = np.zeros_like(x)
print(zeros_like_x)
```

2.4 arange

Another useful kind of generated array is a range of numbers, increasing over a span.

We can create a vector of increasing values using `arange` (**a**rray **range**), which works like the built in Python function `range` does, but returns an 1D array (a vector) instead of a list.

`np.arange()` takes one to three parameters: * `np.arange(end)` – returns a vector of numbers 0..end-1 * `np.arange(start, end)` – returns a vector of numbers start..end-1 * `np.arange(start, end, step)` –returns a vector of numbers start..end-1, incrementing by step (which may be **negative** and/or **fractional**!)

[ ]:
```
np.arange(10)  # [0-10) (excluding 10)
```

[ ]:
```
np.arange(2, 8) # 2,3,4,5,6,7 (no 8 -- the right endpoint is never included!)
```

[ ]:
```
np.arange(0,10,2) # even numbers 0-10
```

[ ]:
```
np.arange(10,0,-2) # even numbers backwards stopping at *2*
# (be careful and remember the right endpoint rule!)
```

[ ]:
```
np.arange(0,10,0.25) # 0 to 10, incrementing by 0.25
```

[ ]:
```
np.arange(0.3, 10.21, 0.95) # all points can be fractional
```

2.4.1 Loading and saving arrays

`np.loadtxt(fname)` and `np.savetxt(arr, fname)` are simple functions to load and save single 1D or 2D arrays as ordinary text files.

[ ]:
```
sunspots = np.loadtxt("../data/sunspots.csv", delimiter=',') # comma separated
print(sunspots.shape, sunspots.dtype)
```

[ ]:
```
# write to a file, then see what is in it
# a new, blank array
x = np.array([[1,2,3,4], [5,6,7,8]])
# write to "test_array.txt"
np.savetxt("test_array.txt", x)

# print out the contents of the file
# will be space separated by default, in scientific notation
print("test_array.txt")
# this is just the raw contents of the file
with open("test_array.txt") as f:
    print((f.read()))
```

2.5 Random arrays

We can also generate random numbers to fill arrays. Many algorithms use arrays of random numbers as their basic "fuel".

Useful random array functions include: * `np.random.randint(a,b,shape)` creates an array with uniform random *integers* between a and (excluding) b * `np.random.uniform(a,b,shape)` creates an array with uniform random *floating point* numbers between a and b * `np.random.normal(mean,std,shape)` creates an array with normally distributed random floating point numbers between with the given mean and standard deviation.

[ ]:
```
np.random.randint(1,7,(5,5)) # 5x5 array o numbers from 1 through 6 (e.g. dice␣
↪rolls)
```

[ ]:
```
np.random.uniform(0,1,(3,)) # 3 element vector of values from 0-1
```

[ ]:
```
# 3 element vector of values, normally distributed with mean 0, standard␣
↪deviation 1.0
np.random.normal(0,1,(3,))
```

2.6 Joining and stacking

We can also join arrays together. But unlike simple structures like lists, we have to explicitly state on which **dimension** we are going to join.

2.6.1 concatenate and stack

Because arrays can be joined together along different axes, there are two distinct kinds of joining: * We can use `concatenate` to join along an *existing* dimension; * or `stack` to stack up arrays along a *new dimension*.

[ ]:
```
x = np.array([1,2,3,4])
y = np.array([5,6,7,8])
print("Stacked two 1D -> 2D")
print((np.stack([x,y])))
```

[ ]:
```
print("Concatenated two 1D -> 1D")
print((np.concatenate([x,y])))
```

[ ]:
```
# when we have multiple dimensions, we can specify explicitly
# which axis to join on
x = np.zeros((3,3))
y = np.ones((3,3))
```

[ ]:
```
print("Joined on rows")
print(np.concatenate([x,y], axis=0))  # join on rows
```

[ ]:
```
print("Joined on columns")
print(np.concatenate([x,y], axis=1))  # join on columns
```

[ ]:
```
print("Joined by stacking")
print(np.stack([x,y]))
```

2.7 Tiling

We often need to be able to **repeat** arrays. This is called **tiling** and `np.tile(a, tiles)` will repeat `a` in the shape given by `tiles`, joining the result together into a single array.

[ ]:
```
eye = np.array([[1.,0.], [0.,1]])
print(eye)
```

[ ]:
```
print("Repeated 3 times, columns")
np.tile(eye, (1,3))
```

[ ]:
```
print("Repeated 3 times, rows")
np.tile(eye, (3,1))
```

[ ]:
```
print("Repeated 2x2 times")
np.tile(eye, (2,2))
```

3 Worked example

3.0.1 Tabular Data

Many data can be naturally thought of as arrays. For example, a very common structure is a spreadsheet like arrangement of data in tables, with rows and columns. Each row is an **observation**, and each column is a **variable**.

Here we will use a simple dataset, *"Wheat prices in England, 1565-1810"* by **William Playfair**.

We will ask some basic questions about this, and answer them via NumPy. This will introduce slicing and indexing, arithmetic, aggregation/reduction and accumulation. We will also see some basic plotting commands to generate graphs: you do not need to learn these at this point.

[ ]:
```
# the price of wheat 1565-1810 in format:
#       index, year, wheat price, weekly wage
# price: shillings per quarter bushel
# wage: shillings per week

# load a text file, store in the variable wheat, and print it out
wheat = np.loadtxt("../data/Wheat.csv", delimiter=",")
print(wheat)
```

Every array has a **shape** and a **datatype**. We can print these out:

[ ]:
```
print("shape", wheat.shape)
print("dtype", wheat.dtype)
```

This tells us we have a 50 rows x 4 columns array of floating point numbers

We can plot this in a graph:

[ ]:
```
# import plotting
import matplotlib.pyplot as plt
%matplotlib inline

# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)
# plot the wheat price against the year
ax.plot(wheat[:,1], wheat[:,2], label="Wheat price (per quarter bushel)")
# plot the wage of skilled mechanic  against the year
ax.plot(wheat[:,1], wheat[:,3], label="Weekly wage (per week)")
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Shillings")
# add a legend
ax.legend()
```

We have selected pairs of columns to plot; for example `wheat[:,1]` selects the second column, the years. This is an example of slicing.

We can select any subparts of this array using **slice notation**, using square brackets. This is like indexing an array, but we can also select rectangular subregions.

[ ]:
```
print("first row", wheat[0,:]) # : means everything
```

[ ]:
```
print("second column", wheat[:,1]) # note: order is rows then columns
```

This uses **slicing syntax** to select a specific part of the array.

Zero indexing Note that NumPy arrays are indexed starting from `0`!

3.1 Slicing

The slice notation is very simple but powerful. You can give a range specifier for each dimension of the array inside square brackets (in this case we have a 2D array: rows x columns).

A *range specifier* is of the form:

```
start : stop : step
```

Where `start` is the index to start from, `stop` is the end, and `step` is the jump to make between each step. **Any of these parts can be omitted**.

- If `start` is missing, it defaults to 0
- If `end` is missing, it defaults to the last element
- If `step` is missing, it defaults to 1. You don't need to include the second colon if you are omitting step, though it's not an error to do so.

If there is no colon at all (just a number) then the slice takes that element only (start and end are equal). Let's see that in action:

[ ]:
```
print(wheat[5:10, :]) # select the 6th to 11th rows
```

[ ]:
```
print(wheat[:10, 1:3]) # select first ten rows and second and third columns
```

Slices wil give us back arrays. These have shapes, which we can look at

[ ]:
```
print(wheat[:10, 1:3].shape) # a 10 by 2 array
```

We can access specific elements if we don't have any ranges in the slice, only definite numbers:

[ ]:
```
print(wheat[0, -1]) # the first row, last column; a specific number
```

3.2 Arithmetic

We can do arithmetic on arrays in a single command, without any explicit iteration. We could compute a conversion of bushels into modern units, litres:

[ ]:
```
litres_bushels = 36.3687
price_quarter_bushels = wheat[:,2]  # slice
price_litre = (price_quarter_bushels * 4.0) / litres_bushels
# now print out the price in shillings per litre of wheat
print(price_litre)
```

Note that we are performing arithmetic on a whole array at once; that is we apply an operation *elementwise*. Lets' convert this further, to kilos, then plot the graph again:

[ ]:
```
wheat_kg_litre = 0.79 # about 0.79 kg of wheat in every litre
price_kg = price_litre * wheat_kg_litre
```

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)
# plot the wheat price against the year
ax.plot(wheat[:,1], price_kg, label="Wheat price (per kg)")
# plot the wage of skilled mechanic  against the year
ax.plot(wheat[:,1], wheat[:,3], label="Weekly wage (per week)")
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Shillings")
# add a legend
ax.legend()
```

3.3 Elementwise arithmetic

Say we wanted to compute the ratio of the wheat price and the weekly wage; how many kilos of wheat could a skilled mechanic afford? This again is an arithmetic operation, but instead of an operation between a scalar and an array, it's an operation between two arrays of the same size.

This again is applied **elementwise**, each element from one array being paired with a corresponding one from another.

[ ]:
```
# note: an operation between two *arrays*
kg_per_week = wheat[:,3] / price_kg

# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)

# plot the wheat price against the year
ax.plot(wheat[:,1], kg_per_week, label="kg per week")
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Kg of wheat afforded")
# add a legend
ax.legend()
```

3.4 Aggregate operations

Say we want to know how much wheat a skilled mechanic could afford, *on average each week*, between 1650 and 1700. This is an aggregation operation. First we can select that region:

[ ]:
```
# check we can find the right slice of the data
print(wheat[17:27,:])
```

[ ]:
```
# now slice the corresponding part of our kg pricing
prices_1650_1700 = kg_per_week[17:27]
```

We can **sum** this data with `np.sum()`, which is equivalent to putting a + in between each element, and then divide by the number of elements using the `shape`.

[ ]:
```
print(np.sum(prices_1650_1700) / prices_1650_1700.shape[0])
```

NumPy has a built in `mean` function that does the same thing:

[ ]:
```
print(np.mean(prices_1650_1700))
```

3.4.1 Multiple axes

You can specify a list of axes across which to operate when using an aggregate function. This is mainly used with arrays with more than two dimensions.

[ ]:
```
print(np.mean(wheat, axis=0)) # mean across rows, this is the mean for each␣
↪variable, a useful summary
```

[ ]:
```
# it makes no sense to average together prices and years!
print(np.mean(wheat, axis=1)) # mean across columns, this is meaningless here
```

3.5 Finding things: Boolean arrays

This seems quite fragile, in that we had to specify `[17:27]` to select the years we were interested in. We can use Boolean tests to do this more cleanly. We can compare any array with a value or another array (these are just special cases of a more general rule) and get back an array of Booleans:

[ ]:
```
print(wheat[:,1]>=1650)
```

[ ]:
```
print(wheat[:,1]<1700)
```

We can combine these together using logical operators. These are: * `np.logical_not(x)` inverts a Boolean array * `np.logical_and(x,y)` applies *and* to two arrays * `np.logical_or(x,y)` applies *or* to two arrays

[ ]:
```
print(np.logical_and(wheat[:,1]>=1650, wheat[:,1]<1700))
```

How does that help us? One thing we can do is use `np.nonzero()` to convert every `True` to it's corresponding index.

[ ]:
```
print(np.nonzero(np.logical_and(wheat[:,1]>=1650, wheat[:,1]<1700)))
```

And we would see that `[17:27]` is the right slice to use (remember slices are inclusive of the first specifier, and exclusive of the second).

Even better, though, we can **directly index** an array with another array of Booleans; this will pull out only those values where the index array is `True`

[ ]:
```
selected_years = np.logical_and(wheat[:,1]>=1650, wheat[:,1]<1700) # a Boolean␣
↪array
print(kg_per_week[selected_years])  # select all elements of prices_kg where␣
↪selected_years is True
```

Note what we have done here. We have applied a test to one array (the years) and then used that to index a second array, because we know that they both correspond to the same ordering of elements.

Now we can easily find the mean price of wheat for any period we want:

[ ]:
```
# adjust as you wish
start = 1500
stop = 1750
print(np.mean(kg_per_week[np.logical_and(wheat[:,1]>=start, wheat[:,1]<stop)]))
```

3.6 min, max, arg*

What is least affordable wheat price and what is the most affordable wheat price? This is again an aggregation operation that produces a summary result. `np.min` and `np.max` will tell us, taking the maximum or minimum across the whole array.

[ ]:
```
print("Least kg wheat affordable in the period: ", np.min(kg_per_week))
print("Most kg wheat affordable in the period: ", np.max(kg_per_week))
```

Perhaps more importantly: when did that occur? What years were most affordable or least affordable?

There are two very useful operations that can help us: `np.argmax` which returns the **index** at which a maximum is found, and `np.argmin` which returns the index at which a minimum is found.

[ ]:
```
print("Index with most affordable wheat", np.argmax(kg_per_week))
```

The key idea is that we can use this to cross-reference to the years:

[1]:
```
# note the syntax:
# wheat[:,1] selects the "years" column
# [np.argmax(kg_per_week)] selects the element of that column with the index
# that maximises the array kg_per_week
# This only makes sense because kg_per_week is the same size as the years column
print("Most afforadble year", wheat[:,1][np.argmax(kg_per_week)]) # most wheat␣
 ↪afforded
print("Least affordable year", wheat[:,1][np.argmin(kg_per_week)]) # least wheat␣
 ↪afforded
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-1-985dc873de86> in <module>()
      4 # that maximises the array kg_per_week
      5 # This only makes sense because kg_per_week is the same size as the␣
↪years column
----> 6 print("Most afforadble year", wheat[:,1][np.argmax(kg_per_week)]) #␣
↪most wheat afforded
      7 print("Least affordable year", wheat[:,1][np.argmin(kg_per_week)]) #␣
↪least wheat afforded


NameError: name 'wheat' is not defined
```

Let's verify if that looks right, with a graph:

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)

# plot the wheat price against the year
ax.plot(wheat[:,1], kg_per_week, label="kg per week")
ax.axvline(wheat[:,1][np.argmax(kg_per_week)], label="Maximum affordability",␣
 ↪color='C1')
ax.axvline(wheat[:,1][np.argmin(kg_per_week)], label="Minimum affordability",␣
 ↪color='C2')
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Kg of wheat afforded")
# add a legend
ax.legend()
```

What's the **second** least affordable year? We don't have an equivalent `np.argmin()` for second smallest. But there is a more general function that can answer any query of this type.

`np.argsort(x)` returns the indices that would sort `x` (ascending order by default)

[ ]:
```
ordering = np.argsort(kg_per_week)
print(ordering)
```

[ ]:
```
print("Least affordable", kg_per_week[ordering[0]]) # smallest amount afforded
print("Second least affordable", kg_per_week[ordering[1]]) # *second* smallest␣
 ↪amount afforded
print("Year when second least affordable", wheat[ordering[1], 1]) # year when␣
 ↪when  second smallest amount afforded
```

We can use an array of numbers as an index to another array, similarly to how we did for Booleans. If that array of numbers is the result of `argsort`, we will sort the array:

[ ]:
```
print(kg_per_week[ordering])
```

[ ]:
```
# verify this is the same as sorting it
print(np.sort(kg_per_week[ordering]))
```

Now we can print out the years, in order of wheat price, least affordable first:

[ ]:
```
print(wheat[ordering, 1])
```

We could also try plotting the ordering against the year. If our hypothesis was that wheat was getting more affordable, we should see an increasing trend. The ordering gives us the "rank" of each year, in terms of how affordable food was, where 0=least affordable year, 1=second least affordable, etc.

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)

# plot the wheat price against the year
ax.scatter(wheat[:,1], ordering, label="Rank")
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Affordability ranking")
# add a legend
ax.legend()
```

3.7 Slice assignment

What if we didn't *believe* the data from 1650-1700 was accurate? We might want to show a graph that had that portion omitted.

We already know how to find the indices corresponding to the region 1650-1700. We need to be able to somehow set the values in this range so they are not plotted.

There are three things that are useful to know: * **Slicing works in assignments too.** `x[0:5]=0` sets elements 0 to 5 of `x` to zero. * **There is a special value, "not a number" or NaN that can be used to indicate that a point should not be drawn in a graph.** * **We can copy arrays using** `np.array(x)` **or** `x.copy()`

[ ]:
```
# test slice assignment out
x = np.array([1,2,3,4,5])
# remember -2: means "second last" to "the end"
x[-2:] = 50 # set last two elements to 50
print(x)
```

We must be careful that if we *modify* an array, we do not modify something we don't mean to. If we directly modify `kg_per_week`, then every use of `kg_per_week` will have changed. If we want to experiment with different options, we should copy `kg_per_week` and modify the copy.

We can set a whole portion of an array a simple operation:

[ ]:
```
# specify the elements we want to modify
selected_years = np.logical_and(wheat[:,1]>=1600, wheat[:,1]<1700)
# copy the array, to not modify kg_per_week
kg_per_week_copy = kg_per_week.copy()
# now, we set all of the selected years to the special
# value np.na
kg_per_week_copy[selected_years] = np.nan
```

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)


# plot the original data in a dashed line
ax.plot(wheat[:,1], kg_per_week, ls=':')

# plot the wheat price against the year
# this will have a gap where we removed the data
# by setting it to np.nan
ax.plot(wheat[:,1], kg_per_week_copy, label="kg per week")

# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Kg of wheat afforded")
# add a legend
ax.legend()
```

3.8 Accumulation

If a worker bought as much wheat as they could afford, each year, at that year's price, (and never ate any!) how much wheat would they accumulate at each point in their lifetime? Could we make a chart of the wheat hoard accumulated by such a worker?

Assume a worker earns a wage for 50 years. Starting in 1700, say, we could select the relevant portion of wheat afforded:

[ ]:
```
worker_life = np.logical_and(wheat[:,1]>=1700, wheat[:,1]<1750)
kg_per_week[worker_life]
```

We could multiply these weekly figures by 52 to get the wheat per year:

[ ]:
```
kg_per_year = kg_per_week[worker_life] * 52
```

`sum` would give us the total amount of wheat for the whole lifetime. If, however, we wanted to plot a graph of how big the wheat hoard was each year, we would need something slightly different: the **cumulative sum**. In NumPy `np.cumsum` works like `sum` but **returns all the intermediate sums.**

[ ]:
```
print(np.sum(kg_per_year)) # the sum
np.cumsum(kg_per_year)     # the cumulative sum
```

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)

# plot the wheat price against the year
ax.plot(wheat[worker_life,1], np.cumsum(kg_per_year), label="Wheat hoard")
# add some labels
ax.set_xlabel("Year")
ax.set_ylabel("Kg in the wheat hoard")
# add a legend
ax.legend()
```

This is an example of an **accumulation operation**, which applies an operation to a sequence of values and maintains the intermediate values.

3.8.1 The opposite of cumulative sum: differencing

A corresponding operation is `np.diff()`, which computes the running difference. We can use it to plot the *change* in wheat affordability (the difference in wheat afforded from one year to the next):

[ ]:
```
# create a new figure
fig = plt.figure(figsize=(8,4))
ax = fig.add_subplot(1,1,1)

# plot the wheat price against the year (step will make a clearer plot here)
# note that diff will give us one *less* value than we input, so we drop
# the first row of the wheat table using 1: to compensate
ax.step(wheat[1:,1], np.diff(kg_per_week), label="Wheat differential")

ax.set_xlabel("Year")
ax.set_ylabel("Change in kg")
# add a legend
ax.legend()
```

We can see, for example, that it looks like there was a massive drop in the wheat afforded just before 1800, with the average skilled worker getting almost 2 kg less wheat a week than in the previous year.