numpy

November 7, 2018

1 Python for Data Science

numpy

2 numpy

- If you analyze data, a lot of your work will involve numbers / matrices.
- numpy is a data collection optimized for this work
- Without numpy much fewer scientists would have switched to Python
  - Convenience (for matrix operations)
  - Speed
- Many other libraries have adopted and extended the API of numpy
  - scipy
  - pandas

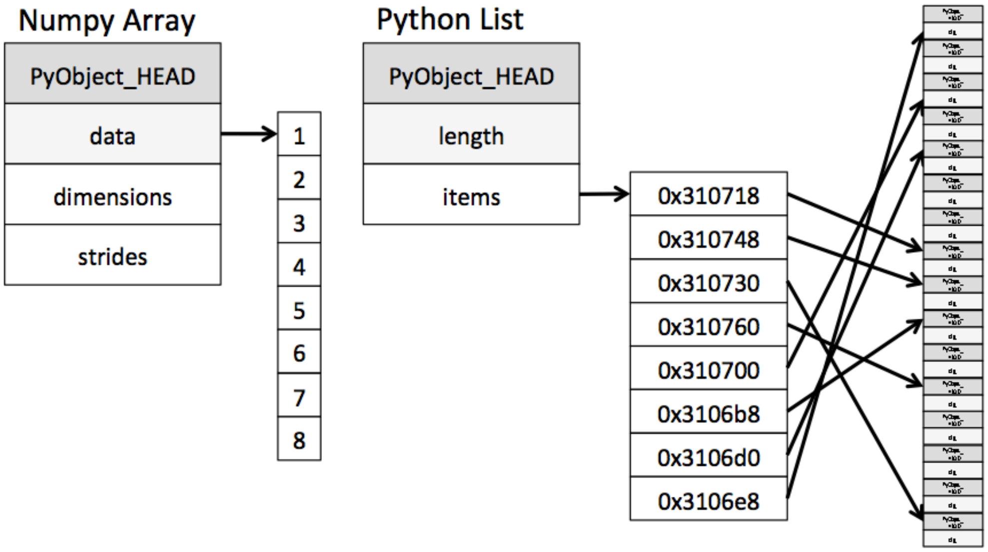
3 numpy is fast

```
In [174]: my_list = list(range(1000000))
          %timeit [x * 2 for x in my_list]
```

```
117 ms ś 18.8 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

```
In [175]: import numpy as np
          np.random.seed(0)  # seed for reproducibility
          my_arr = np.arange(1000000)
          %timeit my_arr * 2
```

```
4.72 ms ś 732 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```



Array Memory Layout

4 Why are `numpy` ndarray faster than lists

Remember how python lists can contain any type:

```
In [3]: L3 = [True, "2", 3.0, 4]
        [type(item) for item in L3]

Out[3]: [bool, str, float, int]
```

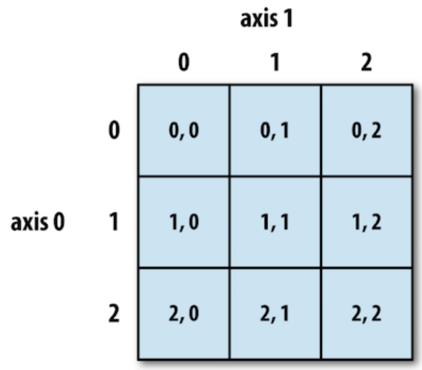
This can be inefficient!

5 `numpy` - Topics For Today

- *Creating arrays*: Various constructors for `numpy` ndarrays
- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Fancy Indexing*: Getting and setting multiple arbitrary elements at once
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*
- *Arithmetic Operations*: +, -, *, ...
- *Aggregation functions*: min, max, sum, ...
- *Loading/Saving*
- *Linear Algebra*: Matrix Multiplication

6 Creating Arrays

```
In [4]: np.array([1, 4, 2, 5, 3])
```



Array indexing

```
Out[4]: array([1, 4, 2, 5, 3])
```

Numpy arrays allow one type only - constructors up-cast types

```
In [5]: np.array([3.14, 4, 2, 3])
```

```
Out[5]: array([3.14, 4.  , 2.  , 3.  ])
```

6.1 Specifying data types at array creation time

```
In [8]: np.array([1, 2, 3, 4], dtype='float32')
```

```
Out[8]: array([1., 2., 3., 4.], dtype=float32)
```

6.2 Creating two-dimensional arrays

```
In [14]: two_d_data = [list(range(i, i + 3)) for i in [2, 4, 6]]
         two_d_data
```

```
Out[14]: [[2, 3, 4], [4, 5, 6], [6, 7, 8]]
```

```
In [15]: np.array(two_d_data)
```

```
Out[15]: array([[2, 3, 4],
                [4, 5, 6],
                [6, 7, 8]])
```

6.3 Creating Arrays from Scratch

6.4 Constant Value Arrays

```
In [16]: np.zeros(10, dtype=int)
```

```
Out[16]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [28]: # Create a 3x5 floating-point array filled with ones
         np.ones((2, 3), dtype=float)

Out[28]: array([[1., 1., 1.],
                [1., 1., 1.]])

In [29]: np.full((3, 4), 3.14)

Out[29]: array([[3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14],
                [3.14, 3.14, 3.14, 3.14]])
```

6.5 Linearly Spaced Values

```
In [19]: # something like builtin range function
         np.arange(0, 20, 2)

Out[19]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [20]: # Create an array of five values evenly spaced between 0 and 1
         np.linspace(0, 1, 5)

Out[20]: array([0.  , 0.25, 0.5 , 0.75, 1.  ])

In [39]: # Create a 3x3 array of values uniformly distributed between 0 and 1
         np.random.random((2, 3))

Out[39]: array([[0.5488135 , 0.71518937, 0.60276338],
                [0.54488318, 0.4236548 , 0.64589411]])

In [40]: # Normally distributed random values, mean 0 and standard deviation 1
         np.random.normal(0, 1, (2, 3))

Out[40]: array([[ 0.95008842, -0.15135721, -0.10321885],
                [ 0.4105985 ,  0.14404357,  1.45427351]])

In [41]: # Create a 3x3 array of random integers in the interval [0, 10)
         np.random.randint(0, 10, (2, 3))

Out[41]: array([[8, 1, 5],
                [9, 8, 9]])
```

6.6 Some Array Data Types

Data type	Description
`bool_`	Boolean (True or False) stored as a byte

Data type	Description
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

6.7 More Array Data Types

- Complex numbers
  - not the most relevant data type for data scientists
- Structured arrays with compound types
  - see e.g. Jake VanderPlas' Python Data Science Handbook chapter

- these types of data are best stored and dealt with in a pandas DataFrame

7 Array Attributes

```
In [2]: x1 = np.random.randint(10, size=6)  # One-dimensional array
        x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
        x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

7.1 Basic Attributes

- `ndim`: the number of dimensions
- `shape`: the size of each dimension
- `size`: the total size of the array
- `dtype`: data type of array elements
- `itemsize`: size (in bytes) of each element
- `nbytes`: size (in bytes) of entire array

```
In [3]: print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
        print("x3 dtype: ", x3.dtype)
        print("x3 itemsize: ", x3.itemsize)
        print("x3 nbytes: ", x3.nbytes)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
x3 dtype:  int64
x3 itemsize:  8
x3 nbytes:  480
```

7.2 Array Indexing and Slicing

Arrays are indexed like this:

`x[index]`

and sliced like this:

`x[start:stop:step]`

With defaults `start=0`, `stop=`*`size of dimension`*, `step=1`.

7.2.1 One Dimensional Arrays

Indexing is just the same as python list indexing:

```
In [7]: x1
```

```
Out[7]: array([4, 3, 6, 9, 8, 1])

In [8]: x1[0]

Out[8]: 4

In [9]: x1[-1]

Out[9]: 1
```

7.2.2 Slicing

You can extract subarrays using standard python List slicing:

```
In [50]: x = np.arange(10)
         x

Out[50]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [51]: x[:5]  # first five elements

Out[51]: array([0, 1, 2, 3, 4])

In [52]: x[5:7] # subarray in the middle

Out[52]: array([5, 6])

In [53]: x[::3] # every third element

Out[53]: array([0, 3, 6, 9])
```

7.2.3 Two Dimensional Arrays

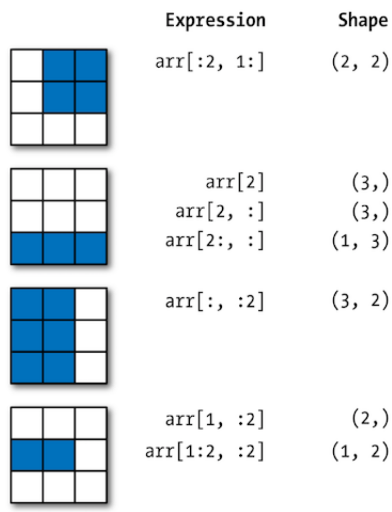
Items can be accessed using a comma-separated tuple of indices:

```
In [54]: x2

Out[54]: array([[1, 5, 3, 9],
                [7, 2, 9, 8],
                [5, 0, 2, 8]])

In [55]: x2[0, 0]

Out[55]: 1

In [56]: x2[2,-1]

Out[56]: 8
```

7.2.4 Slicing Two Dimensional Arrays

Same as with lists and one-dimensional arrays:

```
In [57]: x2

Out[57]: array([[1, 5, 3, 9],
                [7, 2, 9, 8],
                [5, 0, 2, 8]])

In [58]: x2[0,:] # entire first row

Out[58]: array([1, 5, 3, 9])

In [59]: x2[0] # same as x2[0, :]

Out[59]: array([1, 5, 3, 9])

In [60]: x2[:,0] # entire first column

Out[60]: array([1, 7, 5])

In [61]: x2

Out[61]: array([[1, 5, 3, 9],
                [7, 2, 9, 8],
                [5, 0, 2, 8]])

In [62]: x2[:2, :3]  # two rows, three columns

Out[62]: array([[1, 5, 3],
                [7, 2, 9]])

In [63]: x2[:3, ::2]  # all rows, every other column

Out[63]: array([[1, 3],
                [7, 9],
                [5, 2]])

In [64]: x2[::-1, ::-1] # reversing both dimensions

Out[64]: array([[8, 2, 0, 5],
                [8, 9, 2, 7],
                [9, 3, 5, 1]])
```



Array indexing

7.3 Subarrays are Views - Not Copies

Remember: Lists slices are copies

```
In [65]: L = [1,2,3]
         LL = L[:2]
         LL[0] = 99
         print("L: ",L, "\nLL: ", LL)
```

```
L:  [1, 2, 3]
LL:  [99, 2]
```

This is different for subarrays: Those slices are **views**

```
In [66]: a = np.array([1,2,3])
         aa = a[:2]
         aa[0] = 99
         print("a: ", a, "\naa: ", aa)
```

```
a:  [99  2  3]
aa:  [99  2]
```

7.4 Boolean Indexing

You can efficiently index arrays using boolean masks

```
In [116]: grid = np.arange(1, 10)
          grid
```

```
Out[116]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [115]: grid > 5
```

```
Out[115]: array([False, False, False, False, False,  True,  True,  True,  True])
```

```
In [117]: grid[grid > 5]
```

```
Out[117]: array([6, 7, 8, 9])
```

7.5 Fancy Indexing

Can give you arbitrary subsets of arrays given a set of indices

```
In [118]: x = np.random.randint(100, size=10)
          print(x)
```

```
[24 80 66 27 81 70 35 67 65 88]
```

```
In [119]: [x[1], x[4]]
```

```
Out[119]: [80, 81]
```

```
In [123]: ind = [1,4]
          x[ind]
```

```
Out[123]: array([80, 81])
```

8 Reshaping

Often you need to change the shape of an array; this is done with `reshape`

```
In [67]: grid = np.arange(1, 10)
         grid
```

```
Out[67]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [68]: grid = grid.reshape((3, 3))
         grid
```

```
Out[68]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [69]: row_vector = np.arange(3).reshape((1, 3))
         row_vector
```

```
Out[69]: array([[0, 1, 2]])
```

```
In [70]: column_vector = np.arange(3).reshape((3, 1))
         column_vector
```

```
Out[70]: array([[0],
                [1],
                [2]])
```

9 Concatenation

```
In [71]: x = np.array([1, 2, 3])
         y = np.array([3, 2, 1])
         np.concatenate([x, y])

Out[71]: array([1, 2, 3, 3, 2, 1])

In [74]: np.hstack([x, y])

Out[74]: array([1, 2, 3, 3, 2, 1])

In [75]: np.vstack([x, y])

Out[75]: array([[1, 2, 3],
                [3, 2, 1]])
```

10 Splitting

```
In [76]: x = [1, 2, 3, 99, 99, 3, 2, 1]
         x1, x2, x3 = np.split(x, [3, 5])
         print(x1, x2, x3)

[1 2 3] [99 99] [3 2 1]
```

11 Repeated Executions Are Slow In Python

```
In [77]: def compute_reciprocals(values):
             output = np.empty(len(values))
             for i in range(len(values)):
                 output[i] = 1.0 / values[i]
             return output

         values = np.random.randint(1, 10, size=5)
         compute_reciprocals(values)

Out[77]: array([0.33333333, 0.25      , 0.25      , 0.33333333, 0.14285714])

In [80]: big_array = np.random.randint(1, 100, size=1000000)
         %timeit compute_reciprocals(big_array)

2.56 s ś 181 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

12 Fast Vectorized Computations with UFuncs

- Python's defacult implementation CPython is slow for repeated executions
- This is mostly due to the fact that it's not compiled down to bytecode
- Major Bottlenecks are: type-checking and function dispatches
- Various attempts to address this weakness:
  - PyPy, just-in-time compiled implementation of Python
  - Cython, converts Python code to compilable C code
  - Numba, converts snippets of Python code to fast LLVM bytecode.
- None of these have reached broad adoption
- `numpy` is fast since it allows to *vectorize* operations through NumPy's *universal functions* (ufuncs)
- For many tasks you can use plain python and `numpy` for efficient computations

```
In [81]: 1.0 / values # vectorized, fast reciprocal computation with ufuncs

Out[81]: array([0.33333333, 0.25      , 0.25      , 0.33333333, 0.14285714])

In [82]: %timeit (1.0 / big_array)

5.78 ms ś 202 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

12.1 Most arithmetic operations are available as ufuncs

```
In [83]: x = np.arange(4)
         print("x     =", x)
         print("x + 5 =", x + 5)
         print("x - 5 =", x - 5)
         print("x * 2 =", x * 2)
         print("x / 2 =", x / 2)
         print("x // 2 =", x // 2)  # floor division

x     = [0 1 2 3]
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
```

12.2 `ufunc` Arithmetic Operators and Shortcuts

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)

Operator	Equivalent ufunc	Description
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

13 Aggregation Functions

When working with large data sets, aggregations help to understand your data:

- Summing all values
- Min/Max
- Quantiles
- Mean/Median

Again, python itself is slow at that, but `numpy` is fast

```
In [86]: L = np.arange(10)
         L

Out[86]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [87]: sum(L)

Out[87]: 45

In [88]: np.sum(L)

Out[88]: 45

In [95]: big_array = np.random.rand(1000000)
         big_list = big_array.tolist()
         %timeit sum(big_list)
         %timeit np.sum(big_array)

32.1 ms ś 2.97 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
563 ţs ś 22.2 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

Function Name	NaN-safe Version	Description
np.sum	np.nansum	Compute sum of elements
np.prod	np.nanprod	Compute product of elements
np.mean	np.nanmean	Compute mean of elements
np.std	np.nanstd	Compute standard deviation
np.var	np.nanvar	Compute variance

Function Name	NaN-safe Version	Description
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

13.0.1 Multi dimensional aggregates

```
In [100]: M = np.ones((3, 4))
          M

Out[100]: array([[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]])

In [101]: M.sum()

Out[101]: 12.0

In [102]: M.sum(axis=0)

Out[102]: array([3., 3., 3., 3.])

In [103]: M.sum(axis=1)

Out[103]: array([4., 4., 4.])
```

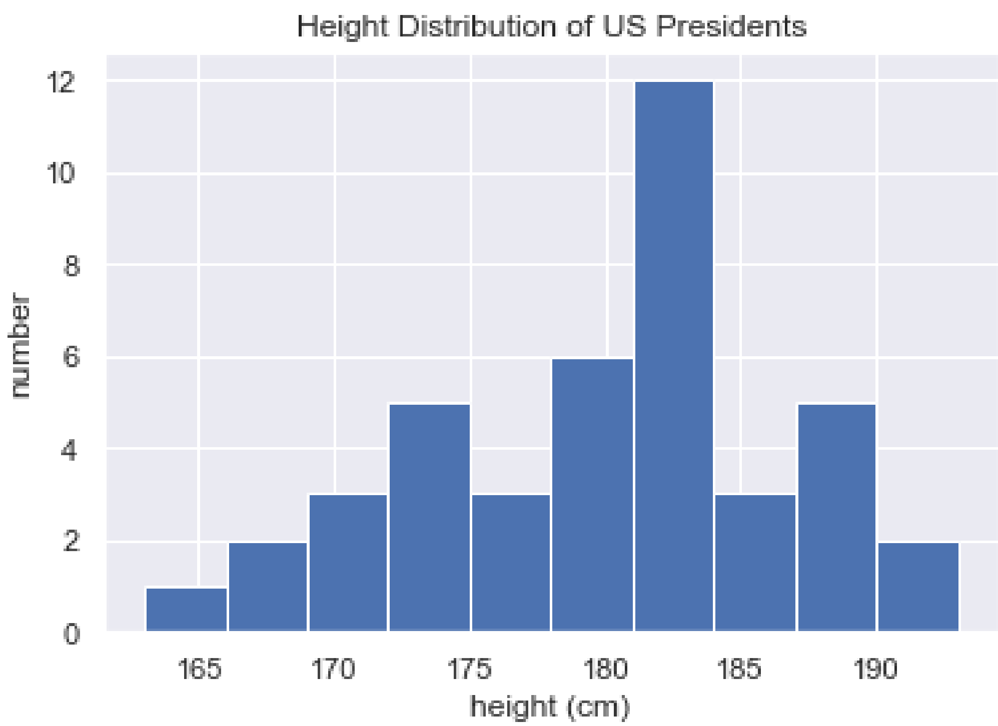
13.1 Example: What is the Average Height of US Presidents?

```
In [127]: heights = np.array([int(line.split(',')[-1]) for line in open("data/president_heights.
          heights[:3]

Out[127]: array([189, 170, 189])

In [111]: print("Mean height:        ", heights.mean())
          print("Standard deviation:", heights.std())
          print("Minimum height:     ", heights.min())
          print("Maximum height:     ", heights.max())

Mean height:        179.73809523809524
Standard deviation: 6.931843442745892
Minimum height:     163
Maximum height:     193
```

```
In [112]: print("25th percentile:   ", np.percentile(heights, 25))
          print("Median:            ", np.median(heights))
          print("75th percentile:   ", np.percentile(heights, 75))
```

```
25th percentile:    174.25
Median:             182.0
75th percentile:    183.0
```

```
In [113]: %matplotlib inline
          import matplotlib.pyplot as plt
          import seaborn; seaborn.set()  # set plot style
          plt.hist(heights)
          plt.title('Height Distribution of US Presidents')
          plt.xlabel('height (cm)')
          plt.ylabel('number');
```



13.2 Fast Sorting in NumPy: `np.sort` and `np.argsort`

```
In [124]: np.sort(heights)
```

```
Out[124]: array([163, 168, 168, 170, 170, 171, 173, 173, 173, 173, 174, 175, 175,
                 177, 178, 178, 178, 178, 179, 180, 182, 182, 182, 182, 183, 183,
                 183, 183, 183, 183, 183, 183, 185, 185, 185, 188, 188, 188, 189,
                 189, 193, 193])
```

```
In [125]: heights[heights.argsort()]

Out[125]: array([163, 168, 168, 170, 170, 171, 173, 173, 173, 173, 174, 175, 175,
                 177, 178, 178, 178, 178, 179, 180, 182, 182, 182, 182, 183, 183,
                 183, 183, 183, 183, 183, 183, 185, 185, 185, 188, 188, 188, 189,
                 189, 193, 193])
```

14 Saving and Loading Arrays

```
In [126]: arr = np.arange(10)
          np.save('some_array', arr)
          loaded_arr = np.load('some_array.npy')
          loaded_arr

Out[126]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

15 Linear Algebra

- For scientists, an important reason to use `numpy` is linear algebra:
  - Matrix arithmetic
  - Matrix multiplication
  - Solving linear systems
  - Matrix decompositions
- We will only talk about the functionalities, not the math concepts
- But we will need all of these tools in later sessions and lectures

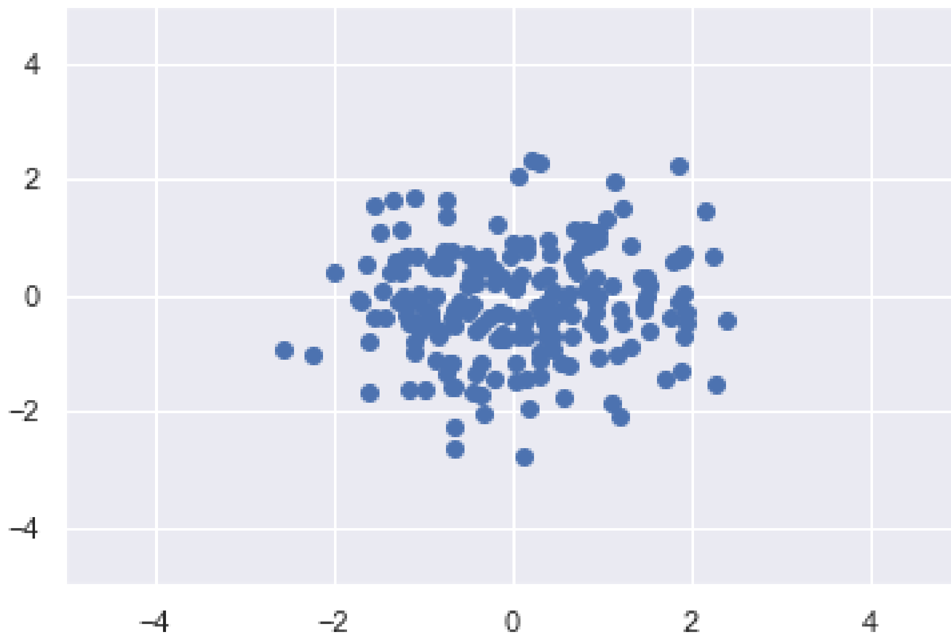
15.1 Matrix Multiplication

```
In [167]: A = np.array([[1,1],[1,1]])
          B = np.array([[2,3],[4,5]])
          A.dot(B)

Out[167]: array([[6, 8],
                 [6, 8]])

In [168]: A @ B

Out[168]: array([[6, 8],
                 [6, 8]])

In [169]: (A @ B).T

Out[169]: array([[6, 6],
                 [8, 8]])
```

15.2 Example: Generating Correlated Gaussian Data

We draw isotropic gaussian variables:
$X \in R^{2,100} \sim \mathcal{N}(0,1)$

In [176]:
```
X = np.random.randn(2,200)
plt.scatter(X[0,:],X[1,:]);plt.xlim([-5,5]);plt.ylim([-5,5]);
```

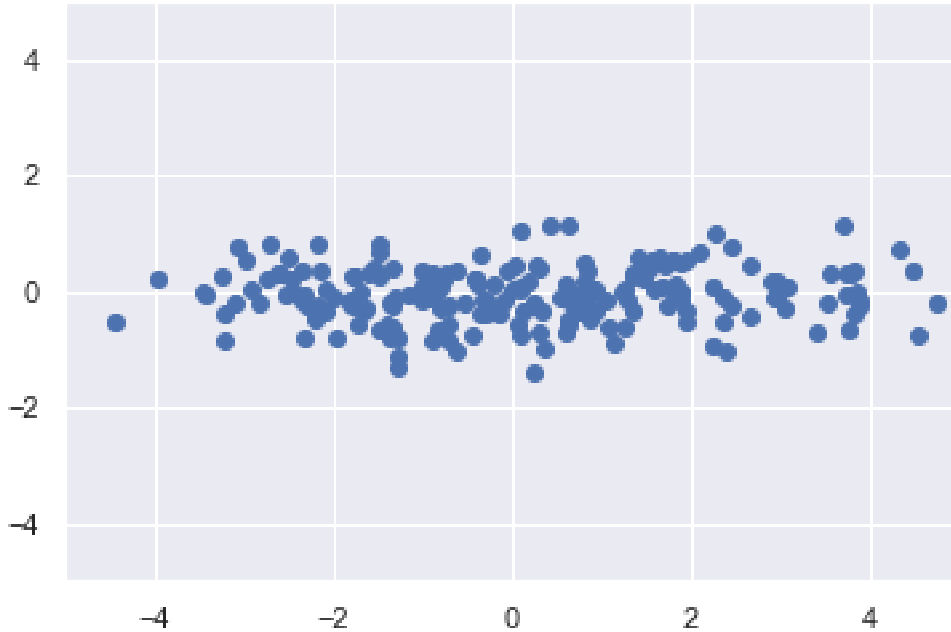


In [177]:
```
# Covariance Matrix
X.dot(X.T)
```

Out[177]:
```
array([[209.66655409,  12.754301  ],
       [ 12.754301  , 181.22473281]])
```

Then we scale the dimensions:
$\begin{bmatrix} 2 & 0 \\ 0 & 0.5 \end{bmatrix} X$

In [178]:
```
S = np.array([[2,0],[0,.5]])
SX = S @ X
plt.scatter(SX[0,:],SX[1,:]);plt.xlim([-5,5]);plt.ylim([-5,5]);
```



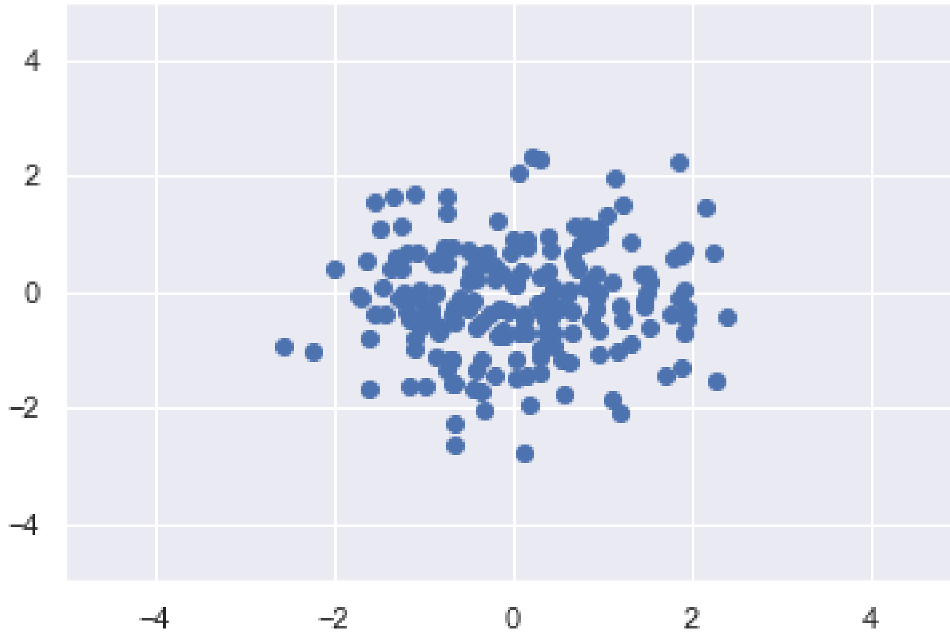
Then we rotate the data by $\theta = 45$ degrees

In [179]:
```
theta = 45
R = np.array([[np.cos(theta),-np.sin(theta)],[np.sin(theta),np.cos(theta)]])
RSX = R @ S @ X
plt.scatter(RSX[0,:],RSX[1,:])
plt.xlim([-5,5]);plt.ylim([-5,5])
```

Out[179]:
```
(-5, 5)
```

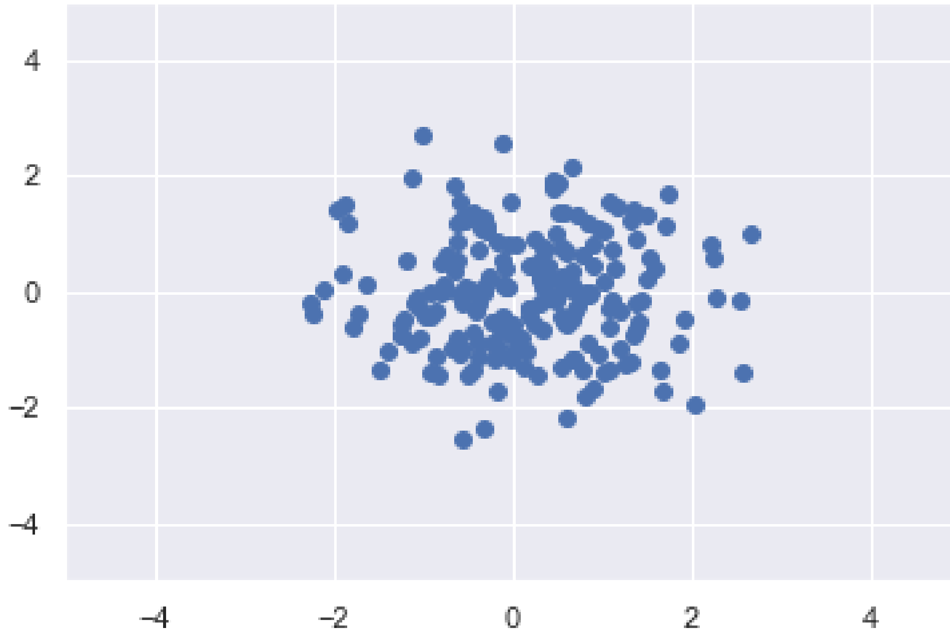
And now back to the original uncorrelated data
$X = (RS)^{-1}RS\,X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} X$

```
In [206]: Xhat = np.linalg.inv(R @ S) @ RSX
          plt.scatter(Xhat[0,:],Xhat[1,:])
          plt.xlim([-5,5]);plt.ylim([-5,5])

Out[206]: (-5, 5)
```

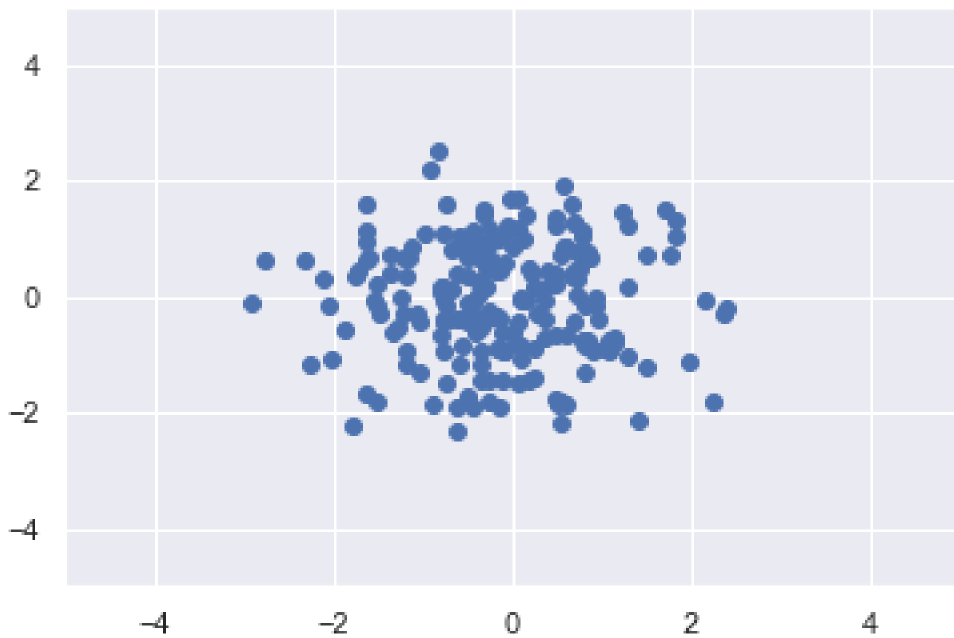


Or with taking the inverse matrix square root of the data
$X = (RSX(RSX)^{\top}))^{-1/2}\,X$

```
In [233]: import scipy
          Xhat = np.linalg.inv(scipy.linalg.sqrtm(RSX @ RSX.T / X.shape[-1])) @ RSX
          plt.scatter(Xhat[0,:],Xhat[1,:])
          plt.xlim([-5,5]);plt.ylim([-5,5])

Out[233]: (-5, 5)
```




```
In [234]: # or with an eigendecomposition
          V,U = np.linalg.eig(RSX @ RSX.T / X.shape[-1])
          Xhat = np.diag(1 / np.sqrt(V)) @ U.T @ RSX
          plt.scatter(Xhat[0,:],Xhat[1,:])
          plt.xlim([-5,5]);plt.ylim([-5,5])

Out[234]: (-5, 5)
```



16 Exercises

All code should be placed in a file `assignment_03.py` with all functions and classes in the top level name space, such that they can be imported by

```
from assignment_03 import *
```

16.1 Creating 1D Arrays

Write a function `assignment_03_01` that creates a one dimensional `numpy` array with the numbers [0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30].

```
def assignment_03_01(...
    return result
```

16.2 Creating 2D Arrays

Write a function `assignment_03_02` that creates a `numpy` array with 4 rows and 5 columns. The content of the array should be 1 for all cells initially. Then, the function should multiply each column by its column index plus 1 and then subtracts its row index.

```
def assignment_03_02(...
    return result
```

16.3 Creating 2D Arrays

Write a function `assignment_03_03` that expects a `numpy` array with 4 rows and 5 columns. Then, extract the bold-faced subarray and return it

.	.	.	.	.
.	**o**	**o**	**o**	.
.	**o**	**o**	**o**	.
.	.	.	.	.

16.4 Manipulating 2D Arrays

Write a function `assignment_03_04` that creates a `numpy` array with 100000 rows and 2 columns. The first column should contain Gaussian distributed variables with mean 1 and standard deviation 2 and the second column should contain Gaussian distributed variables with mean -2 and standard deviation 0.5.