Module 3: NumPy

PUBHLT 0411 · Python for Public Health Data Analysis

Soumik Purkayastha

Dept. of Biostatistics and Health Data Science, University of Pittsburgh

October 22, 2026

Nine micrograms.

Sixty-seven counties and one federal threshold. No loops allowed!

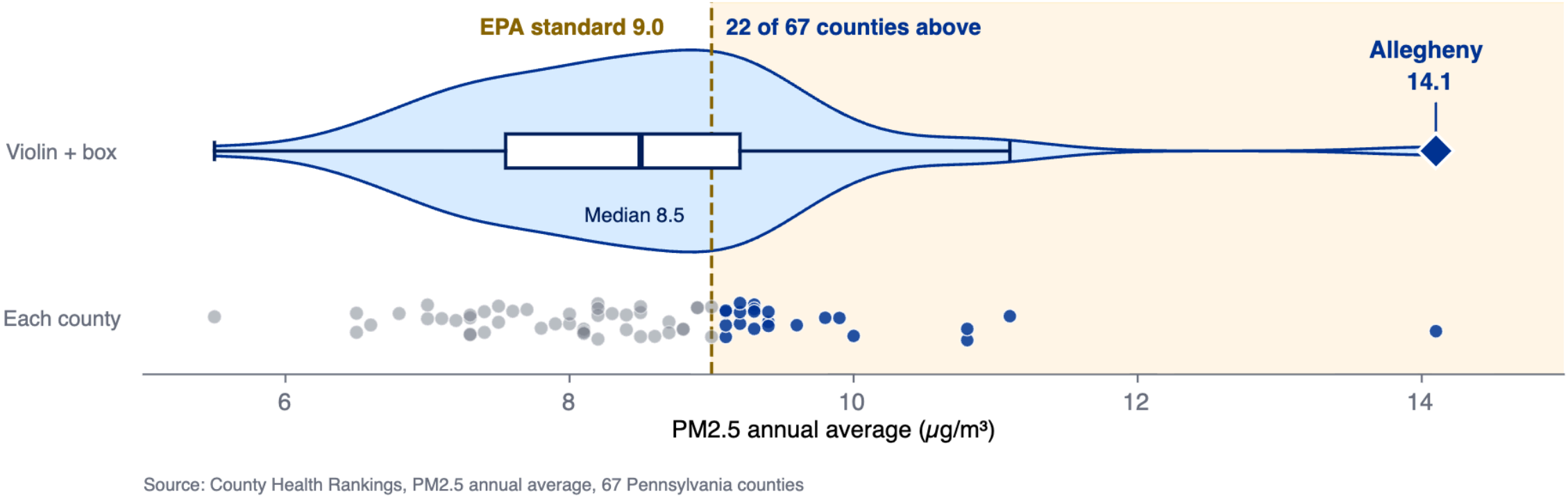
The EPA annual standard for PM2.5 is 9 micrograms per cubic metre

Twenty-two of Pennsylvania's 67 counties exceed it. **One exceeds it by a wide margin.**

> **The question this session opens**
>
> Ranking 67 counties against a threshold, and measuring how far each sits from the state's typical value, requires arithmetic on every county at once.
>
> **This session establishes the array**: one object holding all 67 values, arithmetic that applies to every element in a single expression, and selection by condition rather than by position.

Required from Module 2: `for`, the accumulator pattern, `def` and `return`, and the distinction between counts and rates.

Most counties sit near the standard; Allegheny sits far above it



Source: County Health Rankings, PM2.5 annual average, 67 Pennsylvania counties

Figure 1. PM2.5 annual averages for all 67 Pennsylvania counties, drawn as a violin with a box plot inside and one dot per county below. The median is 8.5 and the middle half runs from 7.5 to 9.2, so the bulk of the state straddles the dashed EPA line at 9.0. Twenty-two counties, in blue, sit above it. Allegheny, at 14.1, sits alone at the right edge.

A NumPy array holds many values of one type

Python Code | Start Over | Run Code

```
1 import numpy as np
2
3 pm25 = np.array([9.1, 14.1, 7.7, 9.9, 8.5])
4
5 print(pm25)
6 print(type(pm25))
7 print(pm25.dtype)
8 print(pm25.shape)
```

1. `np.array()` builds an array from a list
2. `.dtype` reports the single type every element shares
3. `.shape` reports the size along each dimension, as a tuple

`import numpy as np` is the universal convention

Every NumPy example in every textbook and every answer online uses the name `np`. The alias is not required by the language, and departing from it makes code harder for others to read.

An array applies arithmetic to every element at once

A list requires a loop. An array does not.

Python Code | Start Over | Run Code

```python
import numpy as np

pm25 = np.array([9.1, 14.1, 7.7, 9.9, 8.5])

excess = pm25 - 9.0

print(excess)
print(pm25 * 2)
print(pm25 > 9.0)
```

Each operation produces a **new array of the same shape**, with the operation applied element by element. This is **vectorisation**.

> **The same operation on a list does something else entirely**
>
> `[9.1, 14.1] * 2` repeats the list, producing four elements. `np.array([9.1, 14.1]) * 2` doubles each value, producing two. The operator is the same; the type decides the meaning.

Reading a real file into an array

`np.genfromtxt()` reads a CSV directly into arrays. Column 0 holds names; column 1 holds the measurement.

Python Code | Start Over | Run Code

```
import numpy as np

counties = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=0, dtype=str
)
pm25 = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
)

print(pm25.shape)
print(counties[:5])
print(pm25[:5])
```

`skip_header=1` discards the header row. `usecols=` selects a column. Text needs `dtype=str`; numbers are read as `float64` by default.

Source

County Health Rankings, PM2.5 annual average by county, all 67 Pennsylvania counties.

Indexing and slicing follow Module 1's rules

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  pm25 = np.genfromtxt(
4      "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
5  )
6
7  print(pm25[0])
8  print(pm25[1])
9  print(pm25[-1])
10 print(pm25[:5])
11 print(pm25[::10])
```

Zero-indexing, exclusive endpoints, negative indices, and strides are identical to string and list slicing.

A two-dimensional array is indexed `[row, column]`

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  # Overdose deaths per 100,000, four counties × five years (2019–2023)
4  rates = np.array([
5      [46.5, 55.8, 58.5, 56.6, 54.7],   # Allegheny
6      [74.4, 78.5, 82.4, 91.4, 84.8],   # Philadelphia
7      [41.7, 46.6, 39.6, 43.1, 41.7],   # Schuylkill
8      [ 5.1,  8.9,  7.6,  8.9,  4.4],   # Centre
9  ])
10
11 print(rates.shape)
12 print(rates[1, 3])
13 print(rates[2])
14 print(rates[:, 4])
```

`rates[1, 3]` selects one element. `rates[2]` selects a whole row. `rates[:, 4]` selects a whole column — the colon means “every row.”

Rows and columns carry meaning

Here a row is a county and a column is a year. The shape `(4, 5)` states that directly: four counties, five years.

`axis=` chooses the direction a summary collapses

Python Code | Start Over | Run Code

```python
import numpy as np

rates = np.array([
    [46.5, 55.8, 58.5, 56.6, 54.7],   # Allegheny
    [74.4, 78.5, 82.4, 91.4, 84.8],   # Philadelphia
    [41.7, 46.6, 39.6, 43.1, 41.7],   # Schuylkill
    [ 5.1,  8.9,  7.6,  8.9,  4.4],   # Centre
])

print(rates.mean())
print(rates.mean(axis=1))
print(rates.mean(axis=0))
```

1. No `axis=` collapses everything to a single number
2. `axis=1` collapses the **columns**, leaving one value per row — a per-county mean
3. `axis=0` collapses the **rows**, leaving one value per column — a per-year mean

> ⚠ **`axis=` names the dimension removed, not the one kept**
>
> `axis=0` runs down the rows and returns a value per column. Reading it as “axis 0 means rows, so I get row answers” is the standard mistake, and it produces a plausible array of the wrong length.

Summary statistics are methods on the array

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  pm25 = np.genfromtxt(
4      "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
5  )
6
7  print(f"Mean:     {pm25.mean():.2f}")
8  print(f"Median:   {np.median(pm25):.2f}")
9  print(f"Std dev:  {pm25.std():.2f}")
10 print(f"Min–max:  {pm25.min()} to {pm25.max()}")
11 print(f"75th percentile: {np.percentile(pm25, 75)}")
```

`.mean()`, `.std()`, `.min()`, and `.max()` are methods on the array. `np.median()` and `np.percentile()` are functions taking the array as an argument.

`.std()` divides by *n*, not by *n* − 1

NumPy's default is the population standard deviation. R's `sd()` and most statistics courses use the sample version, which is `pm25.std(ddof=1)`. On 67 counties the two differ by under one percent, but the default is not the one a statistics course teaches.

A boolean mask selects elements by condition

A comparison on an array produces an **array of booleans**. Indexing with that array returns the elements where it is `True`.

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  pm25 = np.array([9.1, 14.1, 7.7, 9.9, 8.5])
4
5  over_standard = pm25 > 9.0
6
7  print(over_standard)
8  print(pm25[over_standard])
9  print(over_standard.sum())
```

`.sum()` on a boolean array counts the `True` values, because `True` equals 1.

The mask replaces the counting loop

Module 2 counted with a loop, an `if`, and a counter initialized outside. The mask states the condition once and the count follows from it.

Masking the counties above the standard

Python Code | Start Over | Run Code

```python
import numpy as np

counties = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=0, dtype=str
)
pm25 = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
)

over = pm25 > 9.0

print(f"Counties over 9.0: {over.sum()} of {pm25.size}")
print(f"Share over: {over.mean():.1%}")
print(counties[over][:8])
print(f"Mean where over:  {pm25[over].mean():.2f}")
print(f"Mean where under: {pm25[~over].mean():.2f}")
```

The **same mask** indexes both arrays, because the two are aligned element by element. `~` negates a mask.

⚠ **Masks use `&`, `|`, and `~` — not `and`, `or`, `not`**

`and` and `or` require a single true-or-false value and raise `ValueError` on an array. Each comparison also needs its own parentheses: `(pm25 > 9.0) & (pm25 < 11.0)`.

`np.where()` builds a new array from a condition

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  pm25 = np.array([9.1, 14.1, 7.7, 9.9, 8.5])
4
5  status = np.where(pm25 > 9.0, "Above standard", "At or below")
6  headroom = np.where(pm25 > 9.0, 0.0, 9.0 - pm25)
7
8  print(status)
9  print(headroom)
```

`np.where(condition, value_if_true, value_if_false)` returns a full-length array, choosing element by element.

A mask filters; `np.where()` labels

`pm25[pm25 > 9.0]` returns 22 values. `np.where(pm25 > 9.0, ...)` returns 67, one per county. Selecting and labelling are different jobs.

Standardizing expresses each value in standard deviations

A **z-score** measures distance from the mean in units of the standard deviation.

Python Code | Start Over | Run Code

```
import numpy as np

counties = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=0, dtype=str
)
pm25 = np.genfromtxt(
    "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
)

z = (pm25 - pm25.mean()) / pm25.std()

print(f"Mean of z: {z.mean():.6f}")
print(f"Std of z:  {z.std():.2f}")
print(counties[z > 2])
print(np.round(z[z > 2], 2))
```

The whole transformation is one line, because subtraction and division apply to every element.

A z-score is a statement about this collection

The value describes a county's position among Pennsylvania counties. It carries no information about health effects, and a z-score near zero can still exceed a regulatory standard.

`np.nan` marks a value that is absent

Python Code | Start Over | Run Code

```
1  import numpy as np
2
3  # 2023 overdose rates. Cameron, Juniata, and McKean are suppressed —
4  # fewer than ten deaths, so the state withholds the figure.
5  rates = np.array([54.7, np.nan, 84.8, np.nan, 41.7, np.nan, 4.4])
6
7  print(rates.mean())
8  print(np.isnan(rates))
9  print(np.isnan(rates).sum())
```

Any arithmetic touching `nan` produces `nan`. **One missing value makes the whole mean missing**, which is the safe behaviour: the result announces the gap rather than hiding it.

> ⚠ **`nan` is never equal to anything, including itself**
>
> `np.nan == np.nan` evaluates to `False`. Testing for missingness requires `np.isnan()`.

The `nan`-aware functions skip the gaps

Python Code | Start Over | Run Code

```
1 import numpy as np
2
3 rates = np.array([54.7, np.nan, 84.8, np.nan, 41.7, np.nan, 4.4])
4
5 print(f"mean:       {rates.mean()}")
6 print(f"nanmean:    {np.nanmean(rates):.2f}")
7 print(f"nanmedian:  {np.nanmedian(rates):.2f}")
8 print(f"Reported:   {(~np.isnan(rates)).sum()} of {rates.size} counties")
```

`np.nanmean()`, `np.nanmedian()`, `np.nansum()`, and `np.nanstd()` compute over the present values.

Skipping a value is a decision, and it belongs in the writeup

`np.nanmean()` computes the mean of the counties that reported. Where suppression is concentrated in small rural counties, that mean describes a different population than the one requested, and the count of omitted counties is part of the result.

A slice is a view; modifying it modifies the original

Python Code | Start Over | Run Code

```python
import numpy as np

pm25 = np.array([9.1, 14.1, 7.7, 9.9, 8.5])

first_three = pm25[:3]
first_three[0] = 99.9

print(first_three)
print(pm25)
```

A slice of an array does **not** copy. It is a **view** onto the same memory, and writing through it changes the original array.

> ⚠ **Slicing a list copies; slicing an array does not**
>
> `list[:3]` produces an independent copy. `array[:3]` produces a view. The syntax is identical and the behaviour is not.
>
> **Resolution:** `.copy()` makes an independent array — `first_three = pm25[:3].copy()`.

The outlier.

Putting the session together.

Exercise: standardize and rank

Exercise | Start Over | Show Hint | Run Code

```
1  import numpy as np
2
3  counties = np.genfromtxt(
4      "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=0, dtype=str
5  )
6  pm25 = np.genfromtxt(
7      "data/pa_air_pollution.csv", delimiter=",", skip_header=1, usecols=1
8  )
9
10 z = (pm25 - pm25.______()) / pm25.std()
11
12 worst = z.argmax()
13
14 print(f"Highest PM2.5: {counties[worst]}")
15 print(f"  concentration: {pm25[worst]} ug/m3")
16 print(f"  z-score:       {z[worst]:+.2f}")
17 print(f"Counties over the 9.0 standard: {(pm25 > 9.0).sum()}")
```

Allegheny County

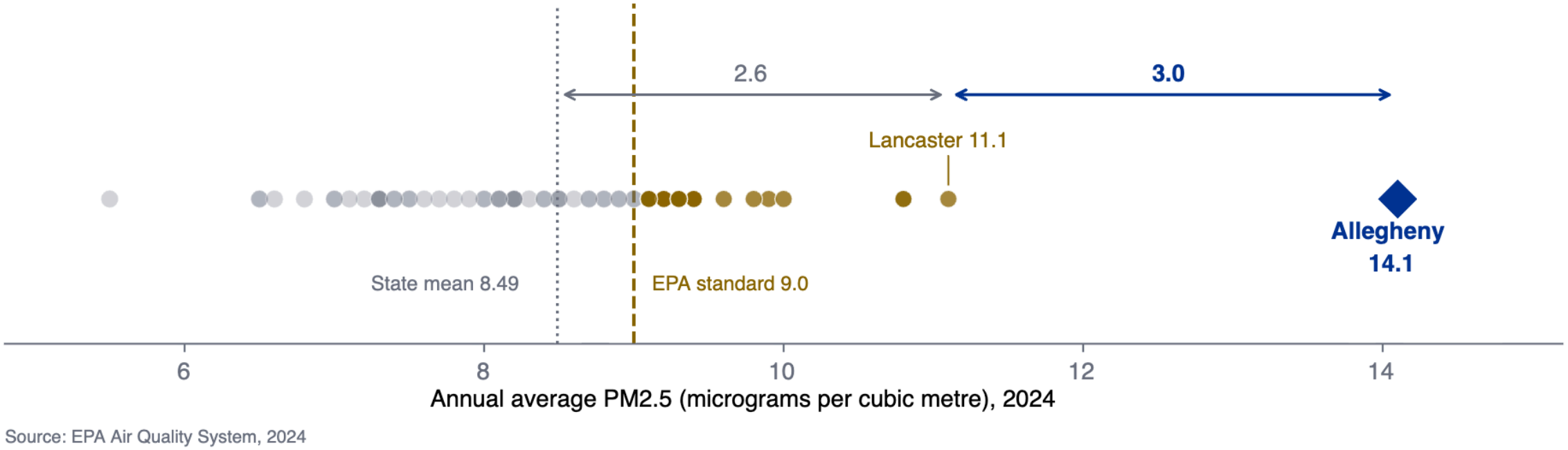


Figure 2. Each dot is one Pennsylvania county's annual average PM2.5. Sixty-six counties fall between 5.5 and 11.1; Allegheny sits alone at 14.1, a blue diamond well to the right of every other county. Twenty-two counties exceed the EPA standard of 9.0, marked in gold, and all but Allegheny cluster between 9.1 and 11.1. Two arrows compare the gaps: 3.0 from Lancaster up to Allegheny, and 2.6 from the state mean of 8.49 up to Lancaster.

Allegheny County's PM2.5 average is 14.1 micrograms per cubic metre — a z-score of +4.37.

The state mean is 8.49. The next-highest county, Lancaster, is 11.1.

What the magnitude means

Challenge: the outlier hides the others

Allegheny inflates the standard deviation for every county. Remove it and recompute.

Exercise | Start Over | Show Hint | Run Code

```
keep = counties ______ "Allegheny"
rest = pm25[keep]
z_rest = (rest - rest.mean()) / rest.______()

top = z_rest.argmax()
print(f"Without Allegheny, highest: {counties[keep][top]}, z = {z_rest[top]:+.2f}")
print(f"Beyond 2 SD, all 67:        {((z > 2) | (z < -2)).sum()}")
print(f"Beyond 2 SD, without it:    {((z_rest > 2) | (z_rest < -2)).sum()}")
```

Your turn!

Lancaster's PM2.5 did not change, yet its z-score rose from +2.03 to +2.47. Why? And why does removing one extreme county produce **more** counties beyond 2 SD, not fewer?

Session outcomes

1. Building arrays with `np.array()`, and reading real ones with `np.genfromtxt()`
2. Inspecting `.dtype`, `.shape`, and `.size`
3. Indexing and slicing in one dimension, and `[row, column]` in two
4. **Vectorised arithmetic** — one expression applied to every element
5. Summarising with `.mean()`, `.std()`, `np.median()`, and `np.percentile()`
6. Collapsing a chosen dimension with **`axis=`**
7. Selecting with a **boolean mask**, and labelling with **`np.where()`**
8. Standardising to **z-scores**
9. Handling `np.nan` with `np.isnan()` and the `nan`-aware functions
10. Distinguishing a **view** from a **copy**

> **Preparation for the lab, Tuesday 10/27**
>
> The lab reads both course files, masks against the standard, standardises overdose rates, and handles the suppressed counties.

Module 4 attaches these arrays to labelled columns and calls the result a DataFrame.

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science **Interactively** at www.DataCamp.com

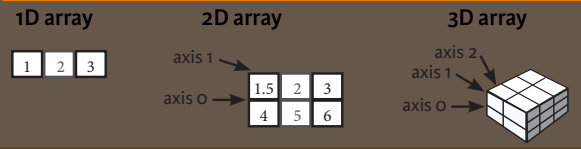
NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))                  Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)  Create an array of ones
>>> d = np.arange(10,25,5)           Create an array of evenly
                                     spaced values (step value)
>>> np.linspace(0,2,9)               Create an array of evenly
                                     spaced values (number of samples)
>>> e = np.full((2,2),7)             Create a constant array
>>> f = np.eye(2)                    Create a 2X2 identity matrix
>>> np.random.random((2,2))          Create an array with random values
>>> np.empty((3,2))                  Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64       Signed 64-bit integer types
>>> np.float32     Standard double-precision floating point
>>> np.complex     Complex numbers represented by 128 floats
>>> np.bool        Boolean type storing TRUE and FALSE values
>>> np.object      Python object type
>>> np.string_     Fixed-length string type
>>> np.unicode_    Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape          Array dimensions
>>> len(a)           Length of array
>>> b.ndim           Number of array dimensions
>>> e.size           Number of array elements
>>> b.dtype          Data type of array elements
>>> b.dtype.name     Name of data type
>>> b.astype(int)    Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                        Subtraction
  array([[-0.5,  0. ,  0. ],
         [-3. , -3. , -3. ]])
>>> np.subtract(a,b)                 Subtraction
>>> b + a                            Addition
  array([[ 2.5,  4. ,  6. ],
         [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)                      Addition
>>> a / b                            Division
  array([[ 0.66666667,  1.        ,  1.        ],
         [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)                   Division
>>> a * b                            Multiplication
  array([[  1.5,   4. ,   9. ],
         [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)                 Multiplication
>>> np.exp(b)                        Exponentiation
>>> np.sqrt(b)                       Square root
>>> np.sin(a)                        Print sines of an array
>>> np.cos(b)                        Element-wise cosine
>>> np.log(a)                        Element-wise natural logarithm
>>> e.dot(f)                         Dot product
  array([[ 7.,  7.],
         [ 7.,  7.]])
```

Comparison

```
>>> a == b                           Element-wise comparison
  array([[False,  True,  True],
         [False, False, False]], dtype=bool)
>>> a < 2                            Element-wise comparison
  array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)             Array-wise comparison
```

Aggregate Functions

```
>>> a.sum()                          Array-wise sum
>>> a.min()                          Array-wise minimum value
>>> b.max(axis=0)                    Maximum value of an array row
>>> b.cumsum(axis=1)                 Cumulative sum of the elements
>>> a.mean()                         Mean
>>> b.median()                       Median
>>> a.corrcoef()                     Correlation coefficient
>>> np.std(b)                        Standard deviation
```

Copying Arrays

```
>>> h = a.view()     Create a view of the array with the same data
>>> np.copy(a)       Create a copy of the array
>>> h = a.copy()     Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()         Sort an array
>>> c.sort(axis=0)   Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

```
>>> a[2]                             Select the element at the 2nd index
  3
>>> b[1,2]                           Select the element at row 1 column 2
  6.0                                (equivalent to b[1][2])
```

Slicing

```
>>> a[0:2]                           Select items at index 0 and 1
  array([1, 2])
>>> b[0:2,1]                         Select items at rows 0 and 1 in column 1
  array([ 2.,  5.])
>>> b[:1]                            Select all items at row 0
  array([[1.5, 2., 3.]])             (equivalent to b[0:1, :])
>>> c[1,...]                         Same as [1,:,:]
  array([[[ 3.,  2.,  1.],
          [ 4.,  5.,  6.]]])
>>> a[ : :-1]                        Reversed array a
  array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2]                           Select elements from a less than 2
  array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]    Select elements (1,0),(0,1),(1,2) and (0,0)
  array([ 4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]    Select a subset of the matrix's rows
  array([[ 4. ,5. , 6. , 4. ],       and columns
         [ 1.5, 2. , 3. , 1.5],
         [ 4. , 5. , 6. , 4. ],
         [ 1.5, 2. , 3. , 1.5]])
```

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)              Permute array dimensions
>>> i.T                              Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel()                        Flatten the array
>>> g.reshape(3,-2)                  Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6))                  Return a new array with shape (2,6)
>>> np.append(h,g)                   Append items to an array
>>> np.insert(a, 1, 5)               Insert items in an array
>>> np.delete(a,[1])                 Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)     Concatenate arrays
  array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))                 Stack arrays vertically (row-wise)
  array([[ 1. ,  2. ,  3. ],
         [ 1.5,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]                       Stack arrays vertically (row-wise)
>>> np.hstack((e,f))                 Stack arrays horizontally (column-wise)
  array([[ 7.,  7.,  1.,  0.],
         [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))           Create stacked column-wise arrays
  array([[ 1, 10],
         [ 2, 15],
         [ 3, 20]])
>>> np.c_[a,d]                       Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3)                   Split the array horizontally at the 3rd
  [array([1]),array([2]),array([3])]  index
>>> np.vsplit(c,2)                   Split the array vertically at the 2nd index
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3.,  2.,  3.],
         [ 4.,  5.,  6.]]])]
```

DataCamp
Learn Python for Data Science Interactively

That’s all folks!

See you on Tuesday!