COMP 5070

Statistical Programming
for
Data Science

Python for Big Data:
Intro Notes on NumPy

University of South Australia

The course at a glance

Statistical Programming

- Python Fundamentals
  - Programming Fundamentals, Control Structures, Functions, Data types.
- **Python for Big Data**
  - Numpy, Scipy, Pandas. jupyter notebook and Notebook. More data types and sources.
- R Fundamentals
  - R programming equivalent. Libraries and advanced modelling equivalent.
- R and Python for Big Data
  - Using R with Python. Data wrangling. Compressed File I/O. Visualisation.
- Advanced Case Studies
  - Introduction to PCA and Clustering.

NumPy in a Nutshell

- Designed to efficiently manipulate large multi-dimensional arrays of arbitrary records with small sacrifices for speed.
- Provides:
  - ndarray a fast and space-efficient multidimensional array providing vectorised arithmetic operations.
  - Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Basic Linear algebra, random number generation, Fourier transform capabilities, tools for integrating code written in C, C++, and Fortran.
- *Does not provide very much high-level data analytical functionality*

Python line magic commands

- jupyter notebook has a selection of special commands called "magic" commands.
- These magics are designed to faciliate common tasks and enable easier control over the jupyter notebook system.
- A line magic command is prefixed by the the percent symbol %.
- Not to be confused with cell magic commands (%%).
- Not all commands are available for both the terminal and notebook version (e.g. %paste). Can also have additional command line options.
- Example: check the execution time of any Python statement using the `%timeit` magic function (available for both terminal and notebook).

```
In [22]: a  = np.random.randn(100, 100)
In [23]: %timeit np.dot(a, a)
10000 loops, best of 3: 402 us per loop
```

4

line vs cell magic commands

- A line magic command (%) and takes as an argument the rest of the line.
- A cell magic command (%%) takes as an argument the rest of the line plus the lines of code below (`shift+enter` signifying the end of the block), e.g.

```
# Line magic command
In [24]: %timeit range(1000)

# Cell magic command
In [24]: %%timeit x=range(10000)
         max(x)
```

- A full list can be found by typing `%lsmagic`
- Note: you might see the following after typing `%lsmagic`

```
Automagic is ON, % prefix IS NOT needed for line magics.
```

- I still recommend using % to avoid developing sloppy habits!

5

The magic %run command

- Can use `%run` to run a .py script from the prompt, e.g.:

```
In [20]: %run script_example.py
```

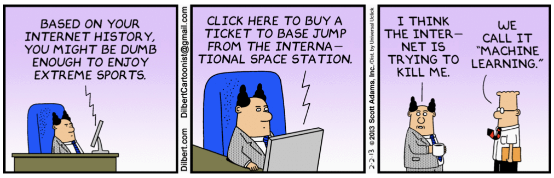
- The script is run in an empty namespace (with no imports or other variables defined) so that the behavior should be identical to running the program on the command line.
- All variables (imports, functions, and globals) defined in the file will then be accessible in the jupyter notebook notebook shell.
- Can use `%run -i` to give a script access to variables already defined in the interactive jupyter notebook namespace

6

Next up …

NumPy basics

- NumPy
- The ndarray
  - creating
  - associated data types
  - indexing and slicing
  - *Ufuncs*
  - vectorisation
  - file I/O
  - data processing



The NumPy ndarray

- An N-dimensional array object, or ndarray.
- Fast, flexible container for large data sets in Python
- Mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

```
In [12]: import numpy as np
         data = np.array([[ 5, -10 , -2],[ 4, 3, 9]])
         type(data)
Out[12]: numpy.ndarray

In [14]: data*10
Out[14]: array([[  50, -100,  -20],
                [  40,   30,   90]])

In [15]: data + data
Out[15]: array([[ 10, -20,  -4],
                [  8,   6,  18]])
```

Avoids the need for for loops by using vectorisation. Huge computational saving!

Vectorisation assumes arrays are the same size. Otherwise you will need to use broadcasting.

np.array() – creating NumPy arrays

- An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same.
- Every ndarray has a shape: a tuple indicating the size of each dimension.
- Every ndarray has a dtype: an object describing the data type of the array.

```
In [16]: data.shape
Out[16]: (2, 3)

In [17]: data.dtype
Out[17]: dtype('int64')
```

np.array() – creating NumPy arrays

- Creating a NumPy array is straightforward – use `array()`.
- This accepts any sequence-like object (including other arrays) and produces an ndarray.

```
In [18]: data1 = [6, 7.5, 8, 0, 1]
         arr1 = np.array(data1)
         arr1

Out[18]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])

In [19]: type(arr1)

Out[19]: numpy.ndarray
```

10

np.array() – creating NumPy arrays

- Another example:

```
In [20]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
         arr2 = np.array(data2)
         arr2

Out[20]: array([[1, 2, 3, 4],
                [5, 6, 7, 8]])

In [21]: arr2.ndim

Out[21]: 2

In [22]: arr2.shape

Out[22]: (2, 4)
```

Number of dimensions in arr2

- Can convert a list of equal-length lists into an ndarray.

11

Creating higher-dimensional NumPy arrays

```
In [29]: np.zeros(10)

Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [30]: np.zeros((3, 6))

Out[30]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: np.empty((2, 3, 2))

Out[31]: array([[[  1.48219694e-323,   2.17456044e-314],
                 [  2.12274051e-314,   2.18017152e-314],
                 [  2.13786376e-314,   2.13786380e-314]],

                [[  1.48219694e-323,   1.48219694e-323],
                 [  2.13786154e-314,   6.95337719e-309],
                 [  2.13786395e-314,   8.34404872e-309]]])

In [32]: np.arange(15)

Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

1D array (vector)

2D array

3D array – shape defined using a tuple

array-valued version of the range() function

12

Array Creation Functions

Function	Description
array	Convert input data to an ndarray. Copies the input data by default.
asarray	Convert input to ndarray, but do not copy if the input is already an ndarray
arange	Like range but returns an ndarray
ones, ones_like	Produce an array of 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.
zeros, zeros_like	Similar to ones and ones_like however produces arrays of 0's
empty, empty_like	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
eye, identity	Create a square N x N identity matrix

Data Types for ndarrays

- The data type or dtype is a special object containing the information the ndarray needs

```
In [40]: arr1 = np.array([1, 2, 3], dtype=np.float64)
         arr2 = np.array([1, 2, 3], dtype=np.int32)
         arr1.dtype

Out[40]: dtype('float64')

In [41]: arr2.dtype

Out[41]: dtype('int32')
```

- Dtypes are one reason why NumPy is powerful and flexible.
- Dtypes map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data and to connect to code written in a low-level language like C or Fortran.

NumPy data types

Type	Description
int8, uint8	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	Signed and unsigned 16-bit integer types
int32, uint32	Signed and unsigned 32-bit integer types
int64, uint64	Signed and unsigned 64-bit integer types
float16	Half-precision floating point
float32	Standard single-precision floating point. Compatible with C float
float64	Standard double-precision floating point. Compatible with C double and Python float object
float128	Extended-precision floating point

NumPy data types

Type	Description
complex64, complex128, complex 256	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	Boolean type storing True and False values
object	Python object type
string_	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

16

NumPy typecasting

- You can explicitly convert or cast an array from one dtype to another using ndarray's astype() method. Will create a copy of the data.

```
In [45]: arr = np.array([1, 2, 3, 4, 5])
         arr.dtype
Out[45]: dtype('int64')

In [46]: float_arr = arr.astype(np.float64)
         float_arr.dtype
Out[46]: dtype('float64')
```

Converting integers to floats

```
In [49]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
         arr
Out[49]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  10.1])

In [48]: arr.astype(np.int32)
Out[48]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Converting floats to integers – note truncation!

17

NumPy typecasting

- Can also convert a list of strings to numbers

```
In [51]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
         type(numeric_strings)
Out[51]: numpy.ndarray

In [52]: numeric_strings.astype(float)
Out[52]: array([  1.25,  -9.6 ,  42.  ])

In [53]: type(numeric_strings)
Out[53]: numpy.ndarray
```

- Can also use another array's dtype to typecast

```
In [54]: int_array = np.arange(10)
         float_array = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
         int_array.astype(float_array.dtype)
Out[54]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

18

NumPy Indexing and Slicing in 1D

- One-dimensional arrays are simple – they appear to act as lists

```
In [57]: arr = np.arange(10)
         arr
Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [58]: arr[5]
Out[58]: 5

In [60]: arr[5:8] = 12
         arr
Out[60]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

The value 12 has been broadcast to arr

- Array slices are views on the original array – i.e. the original array is modified. Why? NumPy has been designed with large data usage in mind, so copies would likely cause memory problems!
- Can copy a slice using `copy()`, e.g. `arr[5:8].copy()`

19

NumPy Indexing and Slicing in 2D

- In two-dimensional arrays, the elements at each index are not scalars but rather one-dimensional arrays. Indexing is as below:

axis 1

		0	**1**	**2**
axis 0	**0**	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

- E.g. `arr2d[0][2]` or `arr2d[0,2]` will access the same element.

20

NumPy Indexing and Slicing in nD (n>2)

- In higher-dimensional arrays, if you omit later indices the returned object will be a lower-dimensional ndarray, consisting of the data along the higher dimensions.
- We can broadcast a value across a dimension (see over).
- If we use 1 index we will obtain an n-1 dimensional ndarray.
- If we use n-1 indices we will obtain a 1-dimensional ndarray.
- In general, if we use k indices (1 <k < n) the output will be an n-k dimensional ndarray.

21

NumPy Indexing and Slicing in nD (n>2)

```
In [64]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
         arr3d

Out[64]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])

In [66]: arr3d.shape

Out[66]: (2, 2, 3)
```

2 x 2 x 3 ndarray

```
In [67]: arr3d[0]
```

n=3, thus specifying 1 index yields an n-1, i.e. 3-1 = 2D narray

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

In [71]: arr3d[0] = 43
         arr3d
```

The value 43 has been broadcast to arr[0]

```
Out[71]: array([[[43, 43, 43],
                 [43, 43, 43]],

                [[ 7,  8,  9],
                 [10, 11, 12]]])
```

22

NumPy Indexing with Slices

- Like one-dimensional objects such as Python lists, ndarrays can be sliced using the syntax we know from regular python:

```
In [75]: arr[1:6]
Out[75]: array([ 1, 2, 3, 4, 64])
```

- Higher dimensional objects give you more options as you can slice one or more axes and also mix integers.

```
In [76]: arr2d
Out[76]:
array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
In [77]: arr2d[:2]
Out[77]:
arr2d([[1, 2, 3], [4, 5, 6]])
```

sliced along axis 0!

	axis 1		
axis 0	1	2	3
	4	5	6
	7	8	9

NumPy Indexing with Slices

- A slice selects a range of elements along an axis.
- Can pass multiple slices just like you can pass multiple indices:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],[5, 6]])
```

	axis 1		
axis 0	1	2	3
	4	5	6
	7	8	9

- Returns the first two rows of axis 0 and all columns after the first.

NumPy Indexing with Slices

- You can mix integer indexes and slices:

```
In [79]: arr2d[1, :2]
Out[79]: arr2d[1][:2]

In [80]: arr2d[2, :1]
Out[80]: arr2d[2][:1]
```

1	2	3
4	5	6
7	8	9

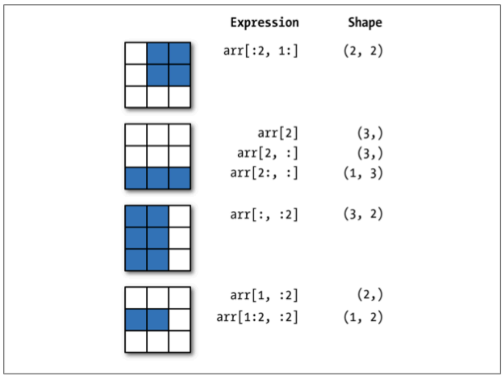
- Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [81]: arr2d[:, :1]
Out[81]: arr2d[:][:1]
```

- Assigning to a slice expression assigns to the whole selection:

```
In [82]: arr2d[:2, 1:] = 0
```

2D Array Slicing Examples



NumPy Boolean Indexing

- Consider the following data:

```
In [82]: names = np.array(['Belinda', 'Malgorzata', 'John',
'Belinda', 'John', 'Jasper', 'Jasper'])

In [83]: colours =
np.array(['green','red','blue','yellow','brown','green','purple
'])

In [84]: names == 'Belinda'
Out[84]: array([ True, False, False,  True, False,
False, False], dtype=bool)
```

these are boolean indices

```
In [85]: colours[names=='Belinda']
Out[85]: array(['green', 'yellow'], dtype='|S6')
```

NumPy Boolean Indexing

- The boolean array must be of the same length as the axis it is indexing.
- Can mix and match boolean arrays with slices or integers (or sequences of integers).

```
In [89]: colours2 =
np.array([['green','red','blue','yellow','brown','green','purpl
e'],['black','pink','pink','brown','white','red','orange']])

In[90]: colours2[0,names == 'Belinda']
Out[90]: array(['green', 'yellow'], dtype='|S6')

In[91]: colours2[1:,names == 'Belinda']
Out[91]: array(['black', 'brown'], dtype='|S6')
```

NumPy Boolean Indexing

- Can also use the operator ! or the negation operator to select everything except …

```
In [92]: colours2[:, names != 'Belinda']
Out[92]: array([['red', 'blue', 'brown', 'green', 'purple'],
    ['pink', 'pink', 'white', 'red', 'orange']], dtype='|S6')


In [93]: colours2[:, -(names=='Belinda')]
Out[93]: array([['red', 'blue', 'brown', 'green', 'purple'],
    ['pink', 'pink', 'white', 'red', 'orange']], dtype='|S6')
```

NumPy Boolean Indexing

- Can also use a mask to combine using & or | as follows:

```
In [94]: mask = (names == 'Belinda') | (names == 'Jasper')
In [95]: mask
Out[95]: array([ True, False, False,True, False, True, True],
dtype=bool)


In [96]: colours2[:,mask]
Out[96]: array([['green', 'yellow', 'green', 'purple'],
       ['black', 'brown', 'red', 'orange']], dtype='|S6')
```

- Note that the keywords and, or do not work with boolean arrays.
- Selecting data from an array by boolean indexing always creates a copy of the data, even if the returned array is unchanged.

NumPy Boolean Indexing

- Can set values using boolean indexing:

```
In [97]: data = np.array([-2,3,1,-4,-12,-3,19])
In [98]: data
Out[98]: array([ -2,   3,   1,  -4, -12,  -3,  19])

In [99]: data[data < 0] = 0
In [100]: data
Out[100]: array([ 0,  3,  1,  0,  0,  0, 19])
```

- Can also set entire rows or columns to a fixed value or set or values or ... the possibilities are endless!
- Or at least they might seem that way!

NumPy Fancy Indexing

- Fancy Indexing is a term used by NumPy – used when we index using integer arrays. Useful to set rows of an array to a specific value, e.g.:

```
In [101]: arr = np.empty((8, 4))
In [102]: for i in range(8):
            arr[i] = i
```

This yields:

```
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

NumPy Fancy Indexing

- Can also select a subset of the rows in a particular order, using either positive or negative indices.
- Note: negative indices selects rows from the end

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]: array(
[[ 4., 4., 4., 4.],
[ 3., 3., 3., 3.],
[ 0., 0., 0., 0.],
[ 6., 6., 6., 6.]])
```

```
In [104]: arr[[-3, -5, -7]]
Out[104]:  array([
[ 5., 5., 5., 5.],
[ 3., 3., 3., 3.],
[ 1., 1., 1., 1.]])
```

selects rows from the end

NumPy Fancy Indexing

- Passing multiple index arrays selects a 1D array of elements corresponding to each tuple of indices:

```
In [105]: arr = np.arange(32).reshape((8, 4))
In [106]: arr
Out[106]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11],
          ...
       [28, 29, 30, 31]])

In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Produces a range of numbers from 0 – 31, which are shaped into an 8x4 array

I.e. the elements (1,0), (5,3), (7,2) and (2,2) were selected! What you were expecting?

NumPy Fancy Indexing

- If you want to select an entire rectangular region you can either:

selects rows   ignores rows and selects columns

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]: array ([[ 4, 7, 5, 6],[20, 23, 21, 22],
                  [28, 31, 29, 30], [ 8, 11, 9, 10]])
```

- Or you can use `np.ix_()`

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]: array ([[ 4, 7, 5, 6],[20, 23, 21, 22],
                  [28, 31, 29, 30], [ 8, 11, 9, 10]])
```

- Fancy indexing also copies the data into a new array.

Universal Functions: Fast Element-wise Array Functions

- A universal function, or *ufunc*, is a function that performs element-wise operations on data in ndarrays.
- Think of them as fast, vectorised wrappers for simple functions that take one or more scalar values and produce one or more scalar results.
- Many ufuncs are simple elementwise transformations, e.g.:

```
In [120]: arr = np.arange(5)
In [121]: np.sqrt(arr)
Out[121]: array([ 0.  , 1.  , 1.4142, 1.7321, 2,  2,2361]
```

Universal Functions: Fast Element-wise Array Functions

- ufuncs that take a single array are known as *unary ufuncs.*
- A *binary ufunc* will take two arrays and return a single array, e.g.

```
In [123]: x = y = np.random.randn(8)
In [124]: np.add(x, y)  # element-wise addition
Out[125]:
array([ 0.267, 0.0974, 0.2002, 0.6117, 0.4655, 0.9222, 0.446, -0.7147])
```

- While not common, a ufunc can return multiple arrays, e.g.

```
In [126]: arr = randn(7) * 5
In [127]: np.modf(arr)  # fractional and integral parts of division
Out[128]:
(array([-0.6808, 0.0636, -0.386 , 0.1393, -0.8806, 0.9363, -0.883 ]),
 array([-2., 4., -3., 5., -3., 3., -6.]))
```

Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`sqrt`	Compute the square root of each element. Equivalent to `arr ** 0.5`
`square`	Compute the square of each element. Equivalent to `arr ** 2`
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
`ceil`	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
`floor`	Compute the floor of each element, i.e. the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is NaN (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-NaN) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise. Equivalent to `-arr`.

Binary ufuncs

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum. `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum. `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `&`, `|`, `^`

Expressing Conditional Logic

- `numpy.where` is the vectorised version of the ternary if expression
- Example:

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
In [142]: cond = np.array([True, False, True, True, False])
```

- Select a value from xarr when cond is true, otherwise select from yarr.

Using List Comprehensions

```
In [143]: result=[(x if c else y)
   .....:       for x, y, c in zip (xarr, yarr, cond)]
```

Using `np.where()`

```
In [145]: result = np.where(cond, xarr, yarr)
In [146]: result
Out[146]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

Expressing Conditional Logic

- Note: The second and third arguments to `np.where()` do not need to be arrays – they can also be scalars.
- Typical use of `np.where()` is to produce a new array of values based on another array.
- Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2.

```
In [147]: arr = np.random.randn(4,4)
Out[148]: array([[ 0.6372, 2.2043, 1.7904, 0.0752],
                 [-1.5926, -1.1536, 0.4413, 0.3483],
                 [-0.1798, 0.3299, 0.7827, -0.7585],
                 [ 0.5857, 0.1619, 1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)
Out[150]: array([[ 2, 2, 2, 2], [-2, -2, 2, 2],
                 [-2, 2, 2, -2],[ 2, 2, 2, -2]])
```

Expressing Conditional Logic: Advanced Example

- Consider this example where I have two boolean arrays, cond1 and cond2, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif
        cond1[i]: result.append(1)
    elif
        cond2[i]: result.append(2)
    else:
        result.append(3)
```

- Could rewrite this as:

```
np.where(cond1 & cond2, 0, np.where(cond1, 1, np.where(cond2, 2,
3)))
```

- Or:

```
result = 1 * cond1 + 2 * cond2 + 3 * -(cond1 | cond2)
```

Array Aggregations

Method	Description
sum	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
mean	Arithmetic mean. Zero-length arrays have NaN mean.
std, var	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
min, max	Minimum and maximum.
argmin, argmax	Indices of minimum and maximum elements, respectively.
cumsum	Cumulative sum of elements starting from 0
cumprod	Cumulative product of elements starting from 1

- Functions such as mean and sum can take an optional axis argument to compute a statistic over the given axis:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data
In [155]: arr.mean(axis=1)   # Column means
Out[155]: array([-1.2833, 0.2844, 0.6574, 0.6743, -0.0187])
```

- *Note!* despite the name, functions cumsum() and cumprod() do not aggregate – they produce an array of intermediate results.

Methods for Boolean Arrays

- Boolean values are coerced to 1 (True) and 0 (False).
- `sum` is thus often used to count True values in a boolean array:

```
In [160]: arr = np.random.randn(100)
In [161]: (arr > 0).sum() # Number of positive values
Out[161]: 44
```

- Two additional methods especially for boolean arrays:
  - *any:* tests whether one or more values in an array is True.
  - *all:* checks if every value is True.

```
In [162]: bools = np.array([False, False, True, False])
In [163]: bools.any() Out[163]: True
In [164]: bools.all() Out[164]: False
```

- Note: *any* and *all* also work with non-boolean arrays, where non-zero elements evaluate to True.

Binary File Input and Output

- NumPy can save and load data either in text or binary format.
- Pandas reads tabular data into memory (later!).
- Binary Files: `np.save` and `np.load` are the two key functions.
- Arrays are saved by default in an uncompressed raw binary format with file extension .npy.

```
In [183]: arr = np.arange(10)
In [184]: np.save('some_array', arr)
```

- If the filename does not end in .npy, the extension will be appended.

Binary File Input and Output

- Can load an np.save array using np.load:

```
In [185]: np.load('some_array.npy')
Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Can save multiple arrays in a zip archive using np.savez and passing the arrays as key-word arguments:

```
In [186]: np.savez('array_archive.npz', a=arr, b=arr)
```

- Loading an .npz file produces a dict-like object:

```
In [187]: arch = np.load('array_archive.npz')
In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

46

Text File Input and Output

- We will predominantly use read_csv and read_table (pandas) however it is always useful to know about NumPy text file I/O.
- np.loadtxt and np.genfromtxt have many options allowing you to specify different delimiters, converter functions for columns, skipping rows, etc.

  Useful for structured arrays/missing data handling
- Example:

This is a .csv file (comma separated values)

```
In [191]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
...
-0.193230,1.047233,0.482803,0.960334

In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')
Out[193]: array([[ 0.5801, 0.1867, 1.0407, 1.1344],
                 [ 0.1942, -0.6369, -0.9387, 0.1241],
                 . . .
                 [-0.1932, 1.0472, 0.4828, 0.9603]])
```

47

Random Number Generation

- numpy.random supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions, e.g.

  4x4 array of random numbers from the Standard Normal Distribution

```
In [208]: samples = np.random.normal(size=(4, 4))
```

- Python's built-in random module only samples one value at a time making numpy.random() well over an order of magnitude faster for generating very large samples, e.g.

```
In [210]: from random import normalvariate
In [211]: N = 1000000

In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

48

Random Number Generation

- Some of the `numpy.random` functions:

Function	Description
seed	Seed the random number generator
permutation	Return a random permutation of a sequence, or return a permuted range
shuffle	Randomly permute a sequence in place
rand	Draw samples from a uniform distribution
randint	Draw random integers from a given low-to-high range
randn	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
binomial	Draw samples a binomial distribution
normal	Draw samples from a normal (Gaussian) distribution
beta	Draw samples from a beta distribution
chisquare	Draw samples from a chi-square distribution
gamma	Draw samples from a gamma distribution
uniform	Draw samples from a uniform [0, 1) distribution

49