Python

Introduction to NumPy

In this lecture

- Difference between Python Lists and NumPy Arrays
- Introduction to NumPy (**Num**erical **Py**thon)
- Creating a NumPy array and type differences
- NumPy Array Functions
- 1D indexing and slicing
- 2D indexing and slicing
- Joining Arrays
- NumPy Arithmetic and Operations
- NumPy Vectorization, Broadcasting and Boolean Masking

Lists in Python

Lists

- A list in Python does use the subscript operator [ ] typically associated with an array. Elements in this list are also indexed.
- The list will maintain a pointer (reference) to objects, rather the integer values (remember Python types are **classes**).
- Lists in python are resizable, unlike static arrays which are fixed.
- Python lists can store elements of different types, whereas arrays are declared to store values of one type.

Lists and Arrays

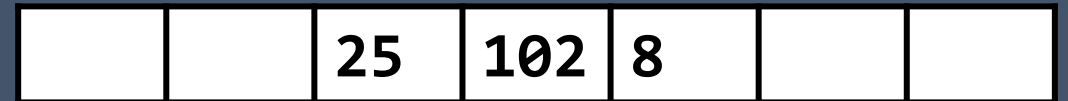## Python List

[27, "Python", 89.13]

		89.13			"Python"			56
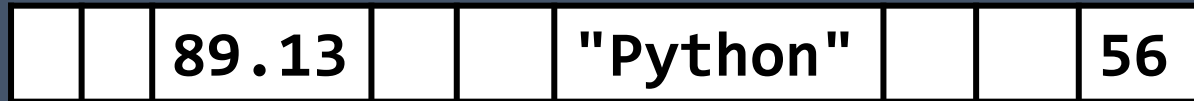
24 bytes   55 bytes   28 bytes

Array

[25, 102, 8]

		25	102	8		

Each block 24 bytes

List

```
In[ ]:   1 |  l = [1,2,3,4,5,6]
         2 |  l
         3 |
```

List

```
In[ ]:  1 |  l = [1,2,3,4,5,6]
        2 |  l
        3 |

Out[]:  [1, 2, 3, 4, 5, 6]
```

List

```
In[ ]:  1 | l = [1,2,3,4,5,6]
        2 | l[0]
        3 |
```

List

```
In[ ]:  1 | l = [1,2,3,4,5,6]
        2 | l[0]
        3 |

Out[]:  1
```

Array

Array

- The items in an array are called **elements**.
- We specify how many elements an array will have when we declare the size of the array **(if 'fixed-size'),** unlike flexible sized collections (ArrayList).
- Elements are numbered and can referred to by number inside the [ ] is called the **index**. This is used when data is input and output.
- Can only store data if it **matches the type** the array is declared with.

Array visualisation

int[] marks = new int[8];

int mark1   28

int mark2   76

int mark3   54

marks[0]	28
marks[1]	76
marks[2]	54
marks[3]	9
marks[4]	27
marks[5]	65
marks[6]	45
marks[7]	17

An Array is a structure that can hold multiple values in individual elements (positions)

Lists vs Arrays

- **List can store data of different types**
- **Array (numpy arrays) store data of the same type**
- We can refer to elements in both a list and array via the [ ]

NumPy

NumPy

- **Num**erical **Py**thon (NumPy) is a package full of methods that can perform useful operations on data.
- NumPy provides a convenient API (Application Programmable Interface) that provides a way to 'interface' with / operate on data.
- It reintroduces types which is more coding but more efficient way to search/sort/store data than the 'loosely' typed nature of Python that we've seen so far.

NumPy

More documentation available at: https://numpy.org

NumPy array

- NumPy arrays are different to Python Lists.
- NumPy arrays reintroduce the 'typed' nature of more 'verbose' languages (C, C++, Java), where everything is explicitly typed.
- NumPy arrays operate like arrays from C and Java where they declared to store data of one type (only integers), unlike Python and JS, which can store data of different types.
- NumPy arrays therefore data is 'cast' – floating point numbers to integers, or in some cases – an error is produced (strings to integers).

Import NumPy

```
In[ ]:   1 | import numpy as np
         2 | np
         3 |
```

Import NumPy

```
In[ ]:   1 |  import numpy as np
         2 |  np
         3 |
```

Out[]: <module 'numpy' from
       '/Users/nick/anaconda3/lib/python3.11/
       site-packages/numpy/__init__.py'>

Same cell

```
In[ ]: 1 | import numpy as np
       2 | a = np.array([1,2,3,4,5,6])
       3 | a
```

Separate cells

```
In[ ]:  1 | import numpy as np
```

```
In[ ]:  1 | a = np.arange(3)
        2 | a
```

np.array

Integer array

```
In[ ]: 1 | a = np.array([1,2,3,4,5,6])
       2 | a
       3 |
```

Integer array

```
In[ ]: 1 | a = np.array([1,2,3,4,5,6])
       2 | a
       3 |
```

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

Floating points

```
In[ ]: 1 | a = np.array([3.14,2,3,4,5])
       2 | a
       3 |
```

Floating points

```
In[ ]:  1 |  a = np.array([3.14,2,3,4,5])
        2 |  a
        3 |
```

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

Notice how the 'integers' have all been upcast to 'floats'!

Floating points

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

Explicit command to upcast the integers to floats

Floating points

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

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

Explicit command to upcast the integers to floats

Array of arrays

```
In[ ]:  1 | a = np.array([[1,2,3],[4,5,6]])
        2 | a
        3 |
```

Array of arrays

```
In[ ]:  1 |  a = np.array([[1,2,3],[4,5,6]])
        2 |  a
        3 |
```

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

Array functions

arange

```
In[ ]:  1 | a = np.arange(3)
        2 | a
        3 |
```

arange

```
In[ ]:  1 | a = np.arange(3)
        2 | a
        3 |
```

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

arange

In[ ]:
```
1 | a = np.arange(10)
2 | a
3 |
```

arange

```
In[ ]:  1 | a = np.arange(10)
        2 | a
        3 |
```

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

zeros

```
In[ ]:   1 | a = np.zeros(6)
         2 | a
         3 |
```

zeros

```
In[ ]:  1 | a = np.zeros(6)
        2 | a
        3 |
```

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

ones

```
In[ ]:  1 | a = np.ones(6)
        2 | a
        3 |
```

ones

```
In[ ]:  1 |  a = np.ones(6)
        2 |  a
        3 |
```

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

Multi-dim

```
In[ ]:  1 | a = np.ones((3, 2))
        2 | a
        3 |
```

Multi-dim

```
In[ ]:  1 | a = np.ones((3, 2))
        2 | a
        3 |
```

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

'other' multi-dim

```
In[ ]:  1 | a = np.full((2, 2), 5)
        2 | a
        3 |
```

'other' multi-dim

```
In[ ]:  1 | a = np.full((2, 2), 5)
        2 | a
        3 |
```

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

'other' multi-dim

```
In[ ]:  1 | a = np.full((3, 3), 7)
        2 | a
        3 |
```

'other' multi-dim

```
In[ ]:   1 | a = np.full((3, 3), 7)
         2 | a
         3 |
```

Out[]: array([[7, 7, 7],
              [7, 7, 7],
              [7, 7, 7]])

eye

```
In[ ]:  1 | a = np.eye(2)
        2 | a
        3 |
```

eye

```
In[ ]:  1 │  a = np.eye(2)
        2 │  a
        3 │
```

Out[]: array([[**1.**, 0.],
              [0., **1.**]])

eye

In[ ]:
```
1 | a = np.eye(3)
2 | a
3 |
```

eye

```
In[ ]:  1 | a = np.eye(3)
        2 | a
        3 |
```

Out[]: array([[**1.**, 0., 0.],
              [0., **1.**, 0.],
              [0., 0., **1.**]])

Random 0-1

```
In[ ]:   1 | a = np.random.random((2,2))
         2 | a
         3 |
```

Random 0-1

```
In[ ]:  1 | a = np.random.random((2,2))
        2 | a
        3 |
```

Out[]: array([[0.0951625, 0.5725122],
              [0.6251428, 0.7182715]])

Random 0 or 1

```
In[ ]:  1 | a = np.random.randint((2,2))
        2 | a
        3 |
```

Random 0 or 1

```
In[ ]:  1 | a = np.random.randint((2,2))
        2 | a
        3 |
```

Out[]: array([1, 0])

Random 1D

```
In[ ]:  1 | a = np.random.randint(2, size=10)
        2 | a
        3 |
```

Random 1D

```
In[ ]:  1 | a = np.random.randint(2, size=10)
        2 | a
        3 |
```

Out[]: array([0, 0, 1, 1, 1, 0, 0, 1, 0, 1])

Random 2D

```
In[ ]:  1 | a = np.random.randint(2, size=(2,2))
        2 | a
        3 |
```

Random 2D

```
In[ ]:  1 | a = np.random.randint(2, size=(2,2))
        2 | a
        3 |
```

Out[]: array([[1, 1],
               [0, 1]])

Random 3 rows

```
In[ ]:  1 │ a = np.random.randint(2, size=(3,2))
        2 │ a
        3 │
```

Random 3 rows

```
In[ ]:  1 | a = np.random.randint(2, size=(3,2))
        2 | a
        3 |
```

Out[]: array([[1, 0],
              [0, 0],
              [1, 0]])

Random 3 cols

```
In[ ]:  1 | a = np.random.randint(2, size=(2,3))
        2 | a
        3 |
```

Random 3 cols

```
In[ ]:  1 | a = np.random.randint(2, size=(2,3))
        2 | a
        3 |
```

Out[]: array([[1, 0, 1],
               [0, 0, 1]])

Random 3x3

```
In[ ]:  1 | a = np.random.randint(3, size=(3,3))
        2 | a
        3 |
```

Random 3x3

```
In[ ]:  1 | a = np.random.randint(3, size=(3,3))
        2 | a
        3 |
```

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

Random 3x3

```
In[ ]: 1 | a = np.random.randint(9, size=(3,3))
       2 | a
       3 |
```

Random 3x3

```
In[ ]:  1 |  a = np.random.randint(9, size=(3,3))
        2 |  a
        3 |
```

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

Random 3D

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3,3))
        2 | a
        3 |
```

Random 3D

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

              [[4, 2, 7],
               [8, 1, 9],
               [1, 0, 2]],

              [[3, 2, 0],
               [8, 2, 6],
               [4, 4, 3]]])
```

Linear Spacing

```
In[ ]:  1 |  a = np.linspace(0, 1, 10)
        2 |  a
        3 |
```

Linear Spacing

```
In[ ]:  1 |  a = np.linspace(0, 1, 10)
        2 |  a
        3 |
```

Out[]: array([0. , 0.11111, 0.22222, 0.33333, 0.44444,
              0.55556, 0.66667, 0.77778, 0.88889, 1.])

10 evenly (linearly) spaced intervals between 0 and 1 (inclusive)

Linear Spacing

```
In[ ]:  1 | a = np.linspace(0, 1, 11)
        2 | a
        3 |
```

Linear Spacing

```
In[ ]:  1 | a = np.linspace(0, 1, 11)
        2 | a
        3 |
```

Out[]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5,
              0.6, 0.7, 0.8, 0.9, 1.])

11 evenly (linearly) spaced intervals between 0 and 1 (inclusive)

Array Attributes

Attributes

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | print("size:", a.size)
        3 | print("shape:", a.shape)
        4 | print("dimensions:", a.ndim)
```

Attributes

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | print("size:", a.size)
        3 | print("shape:", a.shape)
        4 | print("dimensions:", a.ndim)

Out[]:  size: 9
        shape: (3, 3)
        dimensions: 2
```

Attributes

```python
In[ ]:  1 |  a = np.random.randint(9, size=(3,3,3))
        2 |  print("size:", a.size)
        3 |  print("shape:", a.shape)
        4 |  print("dimensions:", a.ndim)
```

Attributes

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3,3))
        2 | print("size:", a.size)
        3 | print("shape:", a.shape)
        4 | print("dimensions:", a.ndim)
```

```
Out[]:  size: 27
        shape: (3, 3, 3)
        dimensions: 3
```

Indexing

- Getting and setting the value of individual array elements:
  x[start : stop : step]

- If any of these are unspecified, they default to the values:
  - start=0,
  - stop=size of dimension,
  - step=1

Indexing

Sample array

```
In[ ]:  1 |  a = np.arange(10)
        2 |  a
        3 |
```

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

Access via index

```
In[ ]:  1 | a = np.arange(10)
        2 | a[3]
        3 |
```

Access via index

```
In[ ]:   1 |  a = np.arange(10)
         2 |  a[3]
         3 |
```

```
Out[]: 3
```

First 3 elements

```
In[ ]: 1 | a = np.arange(10)
       2 | a[:3]
       3 |
```

First 3 elements

```
In[ ]:  1 | a = np.arange(10)
        2 | a[:3]
        3 |
```

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

Elements after 3

```
In[ ]:   1 |  a = np.arange(10)
         2 |  a[3:]
         3 |
```

Elements after 3

```
In[ ]:   1 | a = np.arange(10)
         2 | a[3:]
         3 |
```

Out[]: array([3, 4, 5, 6, 7, 8, 9])

Subarray 3 - 6

```
In[ ]:   1 | a = np.arange(10)
         2 | a[3:6]
         3 |
```

Subarray 3 - 6

```
In[ ]:  1 | a = np.arange(10)
        2 | a[3:6]
        3 |
```

Out[]: array([3, 4, 5])

3-6 step of 2

```
In[ ]:  1 |  a = np.arange(10)
        2 |  a[3:6:2]
        3 |
```

3-6 step of 2

```
In[ ]:  1 |  a = np.arange(10)
        2 |  a[3:6:2]
        3 |
```

Out[]: array([3, 5])

Multiples of 3

```
In[ ]:  1 | a = np.arange(10)
        2 | a[::3]
        3 |
```

Multiples of 3

```
In[ ]:  1 |  a = np.arange(10)
        2 |  a[::3]
        3 |
```

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

Accessing rows and cols

Multi-dim access

- One commonly needed routine is to access a single row or column of an array.
- C and Java require a counter variable to do this manually. This counter increases as one iterates over the array.
- In Python, an easy way to do this is to combine both
  - Indexing [ ]
  - Slicing   :
- NOTE: NumPy array slices return **views** whereas Python list slicing returns **copies** of array data.

Sample array

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a
        3 |
```

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

Select all rows for col 0

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[:,0]
        3 |
```

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

Select all rows for col 0

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[:,0]
        3 |
```

Out[]: array([8, 6, 7])

Select all cols for row 0

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[0,:]
        3 |
```

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

Select all cols for row 0

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[0,:]
        3 |
```

Out[]: array([8, 1, 4])

Select all cols for row 1

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[1,:]
        3 |
```

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

Select all cols for row 1

```
In[ ]:  1 | a = np.random.randint(9, size=(3,3))
        2 | a[1,:]
        3 |
```

Out[]: array([6, 2, 5])

Joining Arrays

Concatenate

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.concatenate([a, b])
        4 | c
```

Concatenate

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.concatenate([a, b])
        4 | c

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

Hstack

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.hstack([a, b])
        4 | c
```

Hstack

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.hstack([a, b])
        4 | c

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

Vstack

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.vstack([a, b])
        4 | c
```

Vstack

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.vstack([a, b])
        4 | c

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

Vstack shape

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.vstack([a, b])
        4 | print(c.shape)
```

Vstack shape

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.vstack([a, b])
        4 | print(c.shape)

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

Arithmetic NumPy

```
Operator        | Equivalent ufunc   |   Output
  5 + 5         |   np.add           |   10
  5 - 4         |   np.subtract      |   1
  5 / 2         |   np.divide        |   2.5
 50 % 4         |   np.mod           |   2    (remainder)
  5 * 10        |   np.multiply      |   50
  5 // 2        |   np.floor_divide  |   2    (round 2.5 down)
  5 ** 2        |   np.power         |   25   (5 x 5)
```

Aggregation functions

Function Name	NaN safe	Description
np.sum	np.nansum	compute sum of elements
np.prod	np.nanprod	compute product of elements
np.median	np.nanmedian	compute median of elements
np.mean	np.nanmean	compute mean of elements
np.std	np.nanstd	compute standard deviation
np.var	np.nanvar	compute variance

Aggregation functions

```
Function Name   |   NaN safe        | Description
 np.min         |   np.nanmin       | find minimum value
 np.max         |   np.nanmax       | find maximum value
 np.argmin      |   np.nanargmin    | find index of min value
 np.argmax      |   np.nanargmax    | find index of max value
 np.percentile  |np.nanpercentile   | rank statistics of elem.
 np.any         |   N/A             | are any elements true
 np.all         |   N/A             | are all elements are true
```

Vectorization

Vectorized

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.sum(a * b)
        4 | print(c)
```

Vectorized

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = np.sum(a * b)
        4 | print(c)

Out[]:  32
```

Non-Vectorized

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = 0
        4 | for i in range(len(a)):
        5 |     c += a[i] * b[i]
        6 | print(c)
```

Non-Vectorized

```
In[ ]:  1 | a = np.array([1,2,3])
        2 | b = np.array([4,5,6])
        3 | c = 0
        4 | for i in range(len(a)):
        5 |     c += a[i] * b[i]
        6 | print(c)

Out[]:  32
```

Broadcasting

Broadcasting

```
In[ ]:  1 | a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 | b = np.array([10,20,30])
        3 | c = a + b
        4 | c
```

Broadcasting

```
In[ ]:  1 |  a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 |  b = np.array([10,20,30])
        3 |  c = a + b
        4 |  c
```

```
Out[]:  array([[11, 21, 31],
               [14, 25, 36],
               [17, 28, 39]])
```

Boolean Masking

Example

```
In[ ]:   1 |  a = np.array([[1,2,3], [4,5,6], [7,8,9]])
         2 |  b = np.array([10,20,30])
         3 |  c = a + b
         4 |  c
```

```
Out[]:  array([[11, 21, 31],
               [14, 25, 36],
               [17, 28, 39]])
```

Boolean Mask

```
In[ ]:  1 | a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 | b = np.array([10,20,30])
        3 | c = a + b
        4 | mask = (c > 20)
        5 | mask
```

Boolean Mask

```
In[ ]:  1 | a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 | b = np.array([10,20,30])
        3 | c = a + b
        4 | mask = (c > 20)
        5 | mask

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

Boolean Mask

```
In[ ]:  1 │ a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 │ b = np.array([10,20,30])
        3 │ c = a + b
        4 │ mask = (c > 20)
        5 │ c[mask]
```

Boolean Mask

```
In[ ]:  1 | a = np.array([[1,2,3], [4,5,6], [7,8,9]])
        2 | b = np.array([10,20,30])
        3 | c = a + b
        4 | mask = (c > 20)
        5 | c[mask]

Out[]: array([22, 33, 25, 36, 28, 39])
```

Challenge 1

Challenge 1

Write a Python script to create a random array with 1000 elements (each element in range [0,10)) and

compute the **average**, **variance**, **standard deviation** of the array elements.

Note: use 1 as random seed.

Documentation: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.random.randint.html

```
Average is 4.413

Variance is 8.216431

Standard deviation is 2.86643175394
```

Answer 1

Answer 1

```python
import numpy as np

np.random.seed(1)

random_array = np.random.randint(10, size=(1000,1))

print('Average is ' + str(np.average(random_array)))
print('Variance is ' + str(np.var(random_array)))
print('Standard deviation is '+ str(np.std(random_array)))
```

Challenge 2

Challenge 2

Write a Python script that:

1. Create two random matrices of integers (each element in range [0,10), size =(3,4))
2. Multiply matrices element wise and save result in new variable
3. Sum each column and save array in new variable.
4. Find maximum value of array
5. Find maximum value index

Note: use 1 as random seed.

```
Maximum value is: 97

Index of Maximum value is: 3
```

Answer 2

```python
import numpy as np

#Create two random matrices of integers (max posible value 10, size = (3,4))
np.random.seed(1) # seed for reproducibility
a = np.random.randint(10, size=(3, 4)) # Two-dimensional array
c = np.random.randint(10, size=(3, 4))

#Multiply matrices element wise
d = np.multiply(a, c)

#Sum Columns
e = np.sum(d,axis=0)

#Fin maximum value
print('Maximum value is: '+str(np.max(e)))

#Find Maximun value index
print('Index of Maximum value is: '+str(np.argmax(e)))
```