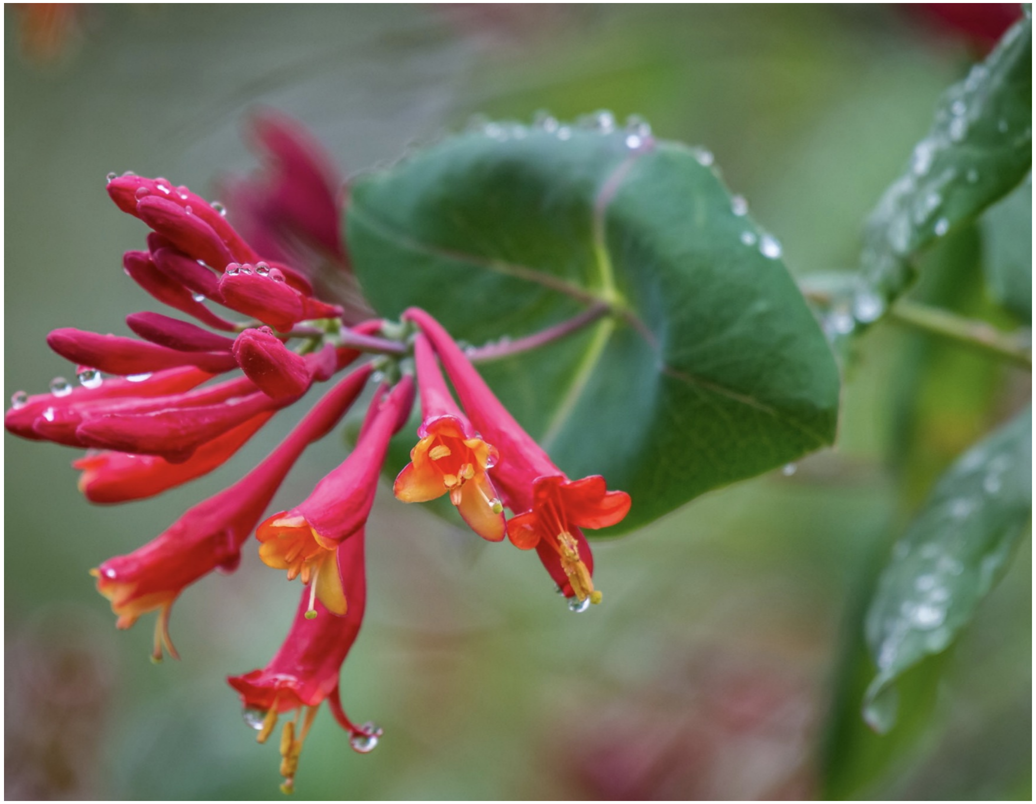
Lecture 20
NumPy



Course: Practical Bioinformatics (BIOL 4220)
Instructor: Michael Landis
Email: michael.landis@wustl.edu

Lecture 20 outline

Last time: genome assembly

This time: NumPy

- NumPy overview
- The ndarray datatype
- NumPy methods

NumPy is a library that improves how Python stores and processes numerical data

Features include:

- Designed for numerical work using ***N-dimensional arrays***
- ***Speed*** and ***memory*** efficiency
- ***Numerical functions*** for standard and NumPy types:
  algebra, trigonometry, linear algebra,
  Fourier analysis, random numbers
- ***Interoperability*** with several popular libraries:
  SciPy, Matplotlib, scikit-learn, pandas, and more

NumPy-compatible Python libraries

NumPy brings the computational power of languages like C and Fortran to Python, a language much easier to learn and use. With this power comes simplicity: a solution in NumPy is often clear and elegant.

Quantum Computing	Statistical Computing	Signal Processing	Image Processing	3-D Visualization	Symbolic Computing	Astronomy Processes	Cognitive Psychology
QuTiP	Pandas	SciPy	Scikit-image	Mayavi	SymPy	AstroPy	PsychoPy
PyQuil	statsmodels	PyWavelets	OpenCV	Napari		SunPy	
Qiskit	Seaborn					SpacePy	

Bioinformatics	Bayesian Inference	Mathematical Analysis	Simulation Modeling	Multi-variate Analysis	Geographic Processing	Interactive Computing
BioPython	PyStan	SciPy	PyDSTool	PyChem	Shapely	Jupyter
Scikit-Bio	PyMC3	SymPy			GeoPandas	IPython
PyEnsembl		cvxpy			Folium	Binder
		FEniCS				

https://numpy.org

NumPy arrays

NumPy implements the ***ndarray*** datatype, which is essentially a lightweight Python wrapper for highly efficient C-arrays

Each *ndarray* is a mutable ***N-dimensional*** container for elements with ***homogenous datatypes***, similar to a list-of-lists

Homogenous dataypes allow compact memory allocation and predictable access times, making *ndarray* operations extremely fast

Example using *ndarray*

```
>>> # import NumPy library
>>> import numpy as np
>>> # create simple 2D array
>>> x = np.array( [[1,2,3], [4,5,6]] )
>>> # list-of-lists converted to array
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> # container type is numpy.ndarray
>>> type(x)
<class 'numpy.ndarray'>
>>> # container dtype is 64-bit integer
>>> x.dtype
dtype('int64')
>>> # element type is 64-bit integer
>>> type(x[0,0])
<class 'numpy.int64'>
```

Create *ndarray* containers

```
>>> np.array( [[1,2],[3,4]] )  # from list-of-lists, 2D
array([[1, 2],
       [3, 4]])
>>> np.empty( [2,3] )          # uninitialized values, 2D
array([[4.9e-324, 9.9e-324, 1.5e-323],
       [2.0e-323, 2.5e-323, 3.0e-323]])
>>> np.zeros([4])              # all zeroes, 1D
array([0., 0., 0., 0.])
>>> np.ones( [2,2,2] )         # all ones, 3D
array([[[1., 1.],
        [1., 1.]],
       [[1., 1.],
        [1., 1.]]])
>>> np.eye(2)                  # identity matrix
array([[1., 0.],
       [0., 1.]])
>>> np.random.rand(2,3)        # random numbers, 0 to 1, 2D
array([[0.03675717, 0.28691042, 0.34546637],
       [0.95096269, 0.78970958, 0.00432774]])
>>> np.arange(0,6,2)           # from 0 to 6, every 2nd value, 1D
array([0, 2, 4])
>>> np.linspace(0,10,5)        # 5 spaced values, from 0 to 10, 1D
array([ 0. , 2.5, 5. , 7.5, 10. ])
```

Get/set *ndarray* dimensions

```
>>> x = np.array( [[1,2,3,4],[5,6,7,8]] )
>>> x                       # array
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> x.T                     # array-transpose
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
>>> x.ndim                  # two dimensions
2
>>> x.shape                 # shape is 2x4 elements
(2, 4)
>>> x.shape = (2,2,2)  # change shape to 2x2x2
>>> x
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
>>> x.flatten()             # return 1D array
array([1, 2, 3, 4, 5, 6, 7, 8])
```

Indexing *ndarray* containers

```
>>> x = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> x
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> x[ 0:2, 1:3 ]             # values by array slicing
array([[2, 3],
       [5, 6]])
>>> x[ [1,2], [0,2] ]         # values by index lists
array([4, 9])
>>> x > 2                     # does element pass test?
array([[False, False, True],
       [ True, True, True],
       [ True, True, True]])
>>> np.where(x > 2)           # elem. indices that pass test
(array([0, 1, 1, 1, 2, 2, 2]), array([2, 0, 1, 2, 0, 1, 2]))
>>> x[x>2]                    # boolean values to index
array([3, 4, 5, 6, 7, 8, 9])
>>> x[ np.where(x > 2) ]      # index tuple to index
array([3, 4, 5, 6, 7, 8, 9])
```

Helper methods for *ndarray*

```
>>> np.append([1,2,3,4], [5,6])     # appends to 1D array
array([1, 2, 3, 4, 5, 6])
>>> np.delete( np.arange(10), 5 )  # deletes elem. w/ value 5
array([0, 1, 2, 3, 4, 6, 7, 8, 9])
>>> a = np.array( [[5,3],[2,1],[7,8],[0,5]] )
>>> a                               # unsorted array
array([[5, 3],
       [2, 1],
       [7, 8],
       [0, 5]])
>>> np.sort(a, axis=0)              # sorts all values
array([[0, 1],
       [2, 3],
       [5, 5],
       [7, 8]])
>>> np.sort(a, axis=1)              # sorts within rows
array([[3, 5],
       [1, 2],
       [7, 8],
       [0, 5]])
```

Merge and split *ndarray* containers

```
>>> a = [[1,2],[3,4]]
>>> b = [[5,6],[7,8]]
>>> x = np.concatenate([a,b], axis=0)   # concatenate by row (axis-0)
>>> x
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> y = np.concatenate([a,b], axis=1)   # concatenate by column (axis-1)
>>> y
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
>>> np.split(x, 2, axis=0)              # split between rows 1,2 (axis-0)
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]])]
>>> np.split(x, 2, axis=1)              # split between cols 1,2 (axis-1)
[array([[1],
        [3],
        [5],
        [7]]),
 array([[2],
        [4],
        [6],
        [8]])]
```

Copy *ndarray* containers

```
>>> x = np.arange(8) # create variable
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> y = x              # shallow copy of x into y
>>> x.shape = (2,4)    # change shape of x
>>> y                  # ... reflected in y
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

shallow copy *shares memory for new variable*

```
>>> x = np.arange(8) # create variable
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> y = np.copy(x)     # deep copy of x into y
>>> x.shape = (2,4)    # change shape of x
>>> y                  # ... y is unchanged
array([0, 1, 2, 3, 4, 5, 6, 7])
```

deep copy *allocates memory for new variable*

Elementwise array operations

```
>>> a = [1, 2]                # standard list
>>> b = [3, 4]                # standard list
>>> x = np.array([1, 2])  # NumPy ndarray
>>> y = np.array([3, 4])  # NumPy ndarray
>>> a * b                     # standard list multiply: fails
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
>>> a + b                     # standard list add: concatenates
[1, 2, 3, 4]
>>> x * y                     # ndarray supports multiply
array([3, 8])
>>> np.multiply(a,b)          # numpy.multiply works with lists
array([3, 8])
```

NumPy arithmetic

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> np.add(a, b)
array([4, 6])
>>> np.subtract(a,b)
array([-2, -2])
>>> np.multiply(a,b)
array([3, 8])
>>> np.divide(a, b)
array([0.33333333, 0.5 ])
>>> np.power(a, b)
array([ 1, 16])
>>> np.power(a, 2)
array([1, 4])
>>> np.mod(a, b)
array([1, 2])
```

NumPy rounding

```
>>> # round up/down to digit
>>> np.around( [1.023, 3.948], 2 )
array([1.02, 3.95])
>>> # round up
>>> np.ceil( [1.023, 3.948] )
array([2., 4.])
>>> # round down
>>> np.floor( [1.023, 3.948] )
array([1., 3.])
```

NumPy summary statistics

```
>>> a = [ 1.9, 2.8, 5.7, 6.6, 8.5 ]
>>> np.sum(a)                   # sum
25.5
>>> np.amin(a)                  # minimum value
1.9
>>> np.amax(a)                  # maximum value
8.5
>>> np.percentile(a, 20)        # 20th percentile
2.6199999999999997
>>> np.median(a)                # 50th percentile
5.7
>>> np.mean(a)                  # mean
5.1
>>> np.var(a)                   # variance
5.94
>>> np.std(a)                   # standard deviation
2.4372115213907883
```

NumPy linear algebra

```
>>> a = np.array( [[1,2],[3,4]] )
>>> b = np.array( [[5,6],[7,8]] )
>>> np.matmul(a,b)                      # matrix-multiply
array([[19, 22],
       [43, 50]])
>>> np.linalg.det(a)                    # matrix determinant
-2.0000000000000004
>>> np.linalg.inv(a)                    # matrix inverse
array([[-2. , 1. ],
       [ 1.5, -0.5]])
>>> np.linalg.solve(a, np.eye(2))       # solves Ax = B
array([[-3., -4.],
       [ 4., 5.]])
>>> np.linalg.eig(a)                    # get eigensystem
(array([-0.37228132, 5.37228132]),      # ... eigenvalues
 array([[-0.82456484, -0.41597356],     # ... eigenvectors
        [ 0.56576746, -0.90937671]]))
>>> np.kron(a,b)                        # Kronecker product
array([[ 5, 6, 10, 12],
       [ 7, 8, 14, 16],
       [15, 18, 20, 24],
       [21, 24, 28, 32]])
```

Overview for Lab 20