
THE UNIVERSITY OF
CHICAGO

# Python Programming

Master's Program in Computer Science

# Python Programming

## Today

Python Scientific Stack: Working with Data

- pip, virtualenv, virtualenvwrapper
- numpy
- pandas
- matplotlib

# Today's Reading

*Python Data Science Handbook*, Jake VanderPlaas

Available electronically via UChicago Library's Safari account:

- Chapter 2: Introduction to NumPy
- Chapter 3: Data Manipulation with Pandas
- Chapter 4: Visualization with Matplotlib

# pip

# PyPI: The Python Package Index

- https://pypi.python.org/pypi
- "A repository of software for the Python programming language."
- As of Saturday evening, there were 122,165 packages there. Each available via **pip**.
- Open to all Python developers for
  - Consumption of other developers' distributions
  - Publication of their own distributions

# PyPI: Popular Packages

- **SciPy**: "A Python-based ecosystem of open-source software for mathematics, science, and engineering"
  - **NumPy**: "The fundamental package for scientific computing with Python"
  - **pandas**: "An open source ... library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language."
  - **matplotlib**: "A 2D plotting library which produces quality figures in a variety of hardcopy formats and interactive environments across platforms."
- **scikit-learn**: "Simple and efficient tools for data mining and data analysis."
- **flask**: "A micro Web development framework for Python."
- **django**: "A high-level Python Web framework that encourages rapid development and clean, pragmatic design."

# pip

A tool for installing Python packages.

```
$ pip install package_name
```

source: https://packaging.python.org/tutorials/installing-packages/

# Virtual Environments: virtualenv

"Python 'Virtual Environments' allow Python packages to be installed in an isolated location rather than being installed globally."

Virtual environments have separate and distinct directories for installed packages.

- **venv**: Available by default in Python 3.3+
- **virtualenv**: Must be installed separately (**pip install virtualenv**)

I use **virtualenv** with a tool called **virtualenvwrapper**.

More info: https://virtualenvwrapper.readthedocs.io source: https://packaging.python.org/tutorials/installing-packages/

# Virtual Environments: virtualenvwrapper

```
$ mkvirtualenv -p /usr/local/bin/python3.6 mpcs
Running virtualenv with interpreter /usr/local/bin/python3.6
Using base prefix '/usr/local'
New python executable in /home/flees/Envs/mpcs/bin/python3.6
Also creating executable in /home/flees/Envs/mpcs/bin/python
Installing setuptools, pip, wheel...done.
virtualenvwrapper.user_scripts creating /home/flees/Envs/mpcs/bin/predeactivate
virtualenvwrapper.user_scripts creating /home/flees/Envs/mpcs/bin/postdeactivate
virtualenvwrapper.user_scripts creating /home/flees/Envs/mpcs/bin/preactivate
virtualenvwrapper.user_scripts creating /home/flees/Envs/mpcs/bin/postactivate
virtualenvwrapper.user_scripts creating /home/flees/Envs/mpcs/bin/get_env_details
```

# Python Scientific Stack

# NumPy

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Types in Python

We have seen that Python's dynamic typing makes it very flexible, but this flexibility comes at a cost.

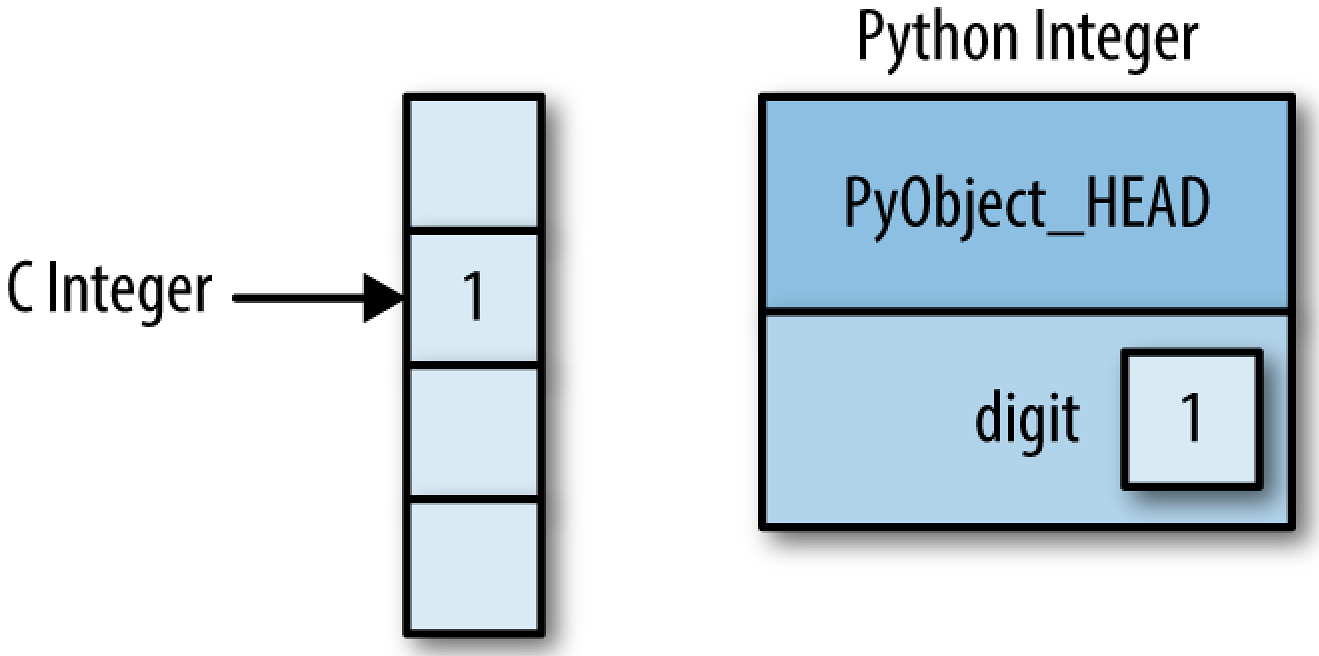
Instances of many of Python's built-in types are "cleverly disguised C structures," containing the data associated with the object, along with header information:

- ob_refcnt: a reference count used for garbage collection
- ob_type: the type of the object
- ob_size: the size of the data members

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Types in Python: Integer

Whereas a C integer is "essentially a label for a position in memory whose bytes encode an integer value," a Python integer is an object in memory containing the object's header info, in addition to the integer value itself.

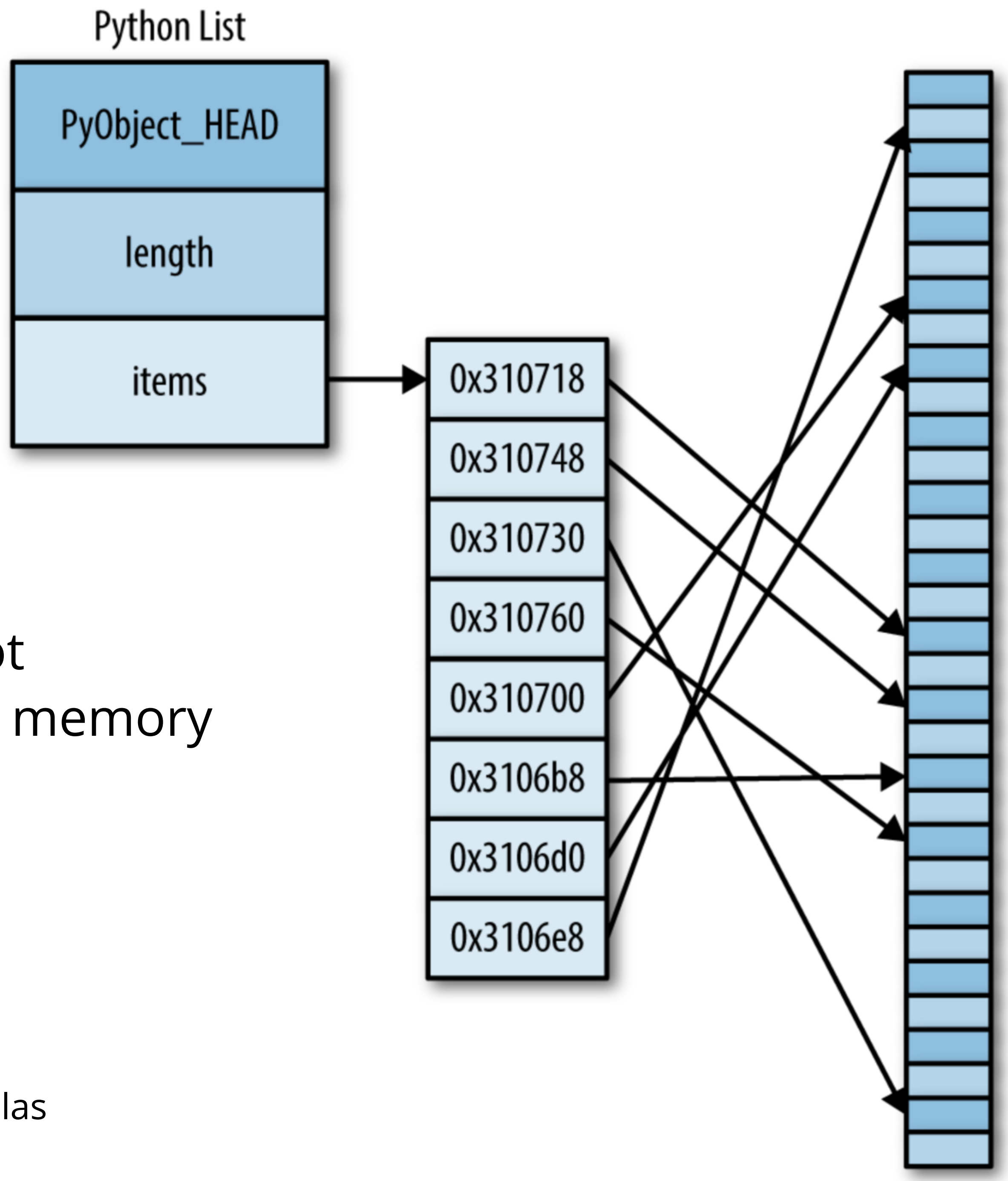


# Data Types in Python: List

Lists are an example of an extremely flexible data type in Python. Aside from the conveniences afforded to us by way of Python's dynamic typing, a Python list can contain elements of heterogeneous types.

In order to accomplish this, each element of a Python list is itself a Python object, complete with all the necessary header information (even if the list happens to contain homogenous elements exclusively).

# Python Programming




Note: The elements of the list are not contiguous (nor are they in order) in memory

Source: *Python Data Science Handbook*, Jake VanderPlas

# Fixed-Type Arrays in Python

When working with homogenous data, Python makes a few alternatives to the **list** type available to us.

The **array** module can be used to create dense, homogenous arrays.

```
>>> import array
>>> ar = array.array('i', range(10))
>>> # the argument 'i' indicates a type.
>>> # see help(array.array) for more
>>> ar
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# Enter NumPy

## Efficient Storage + Efficient Operations: *ndarray*

"While Python's **array** object provides efficient storage of array-based data, NumPy adds to this efficient *operations* on that data."

```
import numpy as np
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# NumPy Arrays

NumPy arrays are constrained to a single type and represent a single contiguous block of data. They lack the flexibility of the Python list, but can be more efficient for storage and manipulation of the data they contain.

NumPy will upcast data where possible for data of different types in a single array. For example, integers can be upcast to floats.

```
>>> import numpy as np
>>> np.array([1, 2, 3, 4.0, 5])
array([ 1.,  2.,  3.,  4.,  5.])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# NumPy Arrays

You can specify the type of the elements of the array using the **dtype** keyword in the **np.array** constructor.

```
>>> import numpy as np
>>> np.array([1, 2, 3, 4, 5], dtype='float32')
array([ 1.,  2.,  3.,  4.,  5.])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# NumPy Arrays

NumPy arrays are multidimensional. The arrays we've seen are simple a special case of an array with just one axis.

- **axes**: In Numpy, dimensions are often called axes.
- **rank**: The number of axes in an ndarray (given by **ndim** attribute).
- **length**: The number of elements in a given axis of an ndarray.
- **shape**: The size of the array in each axis/dimension (represented as a tuple).
- **size**: The total number of elements in the array in all axes.
- **itemsize**: The size in bytes of each element in the array. (Can be specified in the constructor.)
- **data**: The buffer containing the actual elements. Generally not accessed directly.

Source: *Python Data Science Handbook*, Jake VanderPlas

# Multidimensional NumPy Arrays

```
>>> import numpy as np
>>> md = np.array([[1, 2, 3], [4, 5, 6]], dtype='float64')
>>> md[0, 1]
2.0
>>> # row access: index 0 row
>>> md[0,:]
array([ 1.,  2.,  3.])
>>> # column access: index 0 column
>>> md[:,0]
array([ 1.,  4.])
>>> # subset
>>> md[:2,:2]
array([[ 1.,  2.],
       [ 4.,  5.]])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Creating NumPy Arrays

Create multidimensional arrays of zeros or ones.

```
>>> import numpy as np
>>> np.zeros([4, 5])
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones([3, 2])
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

# Creating NumPy Arrays

Create multidimensional array of a single arbitrary value

```
>>> import numpy as np
>>> np.full([7, 4], np.pi)
array([[ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265],
       [ 3.14159265,  3.14159265,  3.14159265,  3.14159265]])
```

# Creating NumPy Arrays

Create an array of a linear sequence, stepping by a given value.

```
>>> import numpy as np
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
```

Create an array of a values between two endpoints, evenly spaced.

```
>>> import numpy as np
>>> np.linspace(0, .5, 5)
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ])
```

# Creating NumPy Arrays

Create an array of uniformly-distributed random values between 0 and 1.

```
>>> import numpy as np
>>> np.random.random((2, 4))
array([[ 0.52540219,  0.49552266,  0.86025628,  0.42584843],
       [ 0.42383527,  0.60370998,  0.4923559 ,  0.00877941]])
```

Create an array of normally-distributed random values with mean 0 and stdev 1

```
>>> import numpy as np
>>> np.random.normal(0, 1, (2, 3))
array([[-0.55781916, -1.20467166,  0.22442593],
       [-1.06398853,  0.36431144,  1.56290961]])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Creating NumPy Arrays

Create an *n* x *n* identity matrix

```
>>> import numpy as np
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# NumPy Array Attributes

```
>>> import numpy as np
>>> a = np.array([range(4), range(4, 8)])
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

```
>>> a.shape
(2, 4)
>>> a.ndim
2
>>> a.size
8
```

```
>>> a.dtype
dtype('int64')
>>> a.nbytes # total size in bytes of array
64
>>> a.itemsize # size in bytes of each element
8
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Reshaping NumPy Arrays

```
>>> import numpy as np
>>> a = np.array([range(4), range(4, 8)])
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a.reshape(4, 2)
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
>>> a.ravel()
array([0, 1, 2, 3, 4, 5, 6, 7])
```

# Reshaping NumPy Arrays

Transpose of ndarray

```
>>> import numpy as np
>>> a = np.array([range(4), range(4, 8)])
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a.T
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Reshaping NumPy Arrays

Modify in place using **resize**. No return value.

```
>>> import numpy as np
>>> a = np.array([range(4), range(4, 8)])
>>> a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> a.resize(4, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

# Concatenate, Stack NumPy Arrays

```
>>> import numpy as np
>>> x = np.array([1, 2, 3])
>>> y = np.array([4, 5, 6])
>>> np.concatenate([x, y])
array([1, 2, 3, 4, 5, 6])
>>> np.vstack([x, y])
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.hstack([x, y])
array([1, 2, 3, 4, 5, 6])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Split NumPy Arrays

```
>>> grid = np.arange(16).reshape((4, 4))
>>> grid
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> upper, lower = np.vsplit(grid, [2])
>>> upper
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> lower
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Split NumPy Arrays

```
>>> grid = np.arange(16).reshape((4, 4))
>>> grid
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> left, right = np.hsplit(grid, [2])
>>> left
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
>>> right
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

# Operations on NumPy Arrays

NumPy provides operations optimized for computation on arrays of data.

"The key to making it fast is to use *vectorized* operations, generally implemented through NumPy's *universal functions* (ufuncs)."

# Operations on NumPy Arrays

Python's flexibility comes with costs. Python has a reputation for slowness in some contexts. Other implementations of the Python interpreter attempt to overcome some of the default implementation's shortcomings (e.g., Cython, PyPy, Numba).

"The relative sluggishness of Python generally manifests itself in situations where many small operations are being repeated—for instance, looping over arrays to operate on each element."

"It turns out that the bottleneck... is not the operations themselves, but the type-checking and function dispatches that CPython must to at each cycle of the loop."

(This is where compiled code has an advantage.)

# Operations on NumPy Arrays

NumPy provides *vectorized* operations via *ufuncs* as an alternative and a way to circumvent bottlenecks of this natures.

Ufuncs' "main purpose is to quickly execute repeated operations on values in NumPy arrays. They are always more efficient than their pure Python loop counterparts, and gain a larger advantage as the arrays grow larger.

***Vectorized* operations**: "Designed to push the loop into the compiled layer that underlies NumPy, leading to much faster execution."

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Array Arithmetic

UFuncs rely on Python's native arithmetic operators. Both unary and binary.

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> -a
array([ 0, -1, -2, -3])
>>> a ** 2
array([0, 1, 4, 9])
>>> a % 2
array([0, 1, 0, 1])
```

```
>>> a + 5
array([5, 6, 7, 8])
>>> a - 5
array([-5, -4, -3, -2])
>>> a * 5
array([ 0,  5, 10, 15])
>>> a / 2
array([ 0. ,  0.5,  1. ,  1.5])
>>> a // 2
array([0, 0, 1, 1])
```

# Array Arithmetic

Operations may be combined in a single expression

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> -(0.5 * a + 1) ** 2
array([-1.  , -2.25, -4.  , -6.25])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Array Arithmetic

Arithmetic operators are wrappers around NumPy functions:

| Operator | ufunc |
|---|---|
| + | np.add |
| - | np.subtract |
| - | np.negative |
| * | np.multiply |
| / | np.divide |
| // | np.floor_divide |
| ** | np.power |
| % | np.mod |

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Additional UFuncs

```
>>> theta = np.linspace(0, np.pi, 3)
>>> theta
array([ 0.        ,  1.57079633,  3.14159265])
>>> np.sin(theta)
array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16])
>>> np.cos(theta)
array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00])
>>> a = np.arange(1, 4)
>>> a
array([1, 2, 3])
>>> np.exp(a) # e^a
array([  2.71828183,   7.3890561 ,  20.08553692])
>>> np.power(3, a) # 3^a
array([ 3,  9, 27])
>>> np.log2(a)
array([ 0.       ,  1.       ,  1.5849625])
>>> np.log10(a)
array([ 0.        ,  0.30103   ,  0.47712125])
```

# Additional UFuncs

More available in **scipy.special**

Source: *Python Data Science Handbook*, Jake VanderPlas

# Aggregation Functions

NumPy has fast, built-in aggregation functions.

```
>>> r = np.random.random(100)
>>> np.sum(r)
52.99375253776082
>>> np.min(r)
0.008102757137900207 2
>>> np.max(r)
0.9979708832198894 7
>>> np.mean(r)
0.5299375253776081 9
>>> np.std(r)
0.2714098716489291 6
>>> np.argmin(r)
78
>>> np.argmax(r)
75
```

```
>>> np.percentile(r, 25)
0.3427250127362183
>>> np.percentile(r, 50)
0.5225954099259119 1
>>> np.percentile(r, 75)
0.7391594966342993 6
>>> np.percentile(r, 100)
0.9979708832198894 7
>>> np.median(r)
0.5225954099259119 1
>>> np.var(r)
0.0736633184284881 95
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Operations on NumPy Arrays

## Broadcasting

Binary operations on arrays of the same size are performed element-wise. *Broadcasting* allows us to perform binary operations on arrays of different sizes. We can think of the operation "broadcasting" the smaller array across the larger array.

We saw this with scalars in the first examples of binary operations on ndarrays. (Think of a scalar as a zero-dimensional array.)

Source: *Python Data Science Handbook*, Jake VanderPlas

# Operations on NumPy Arrays

## Broadcasting

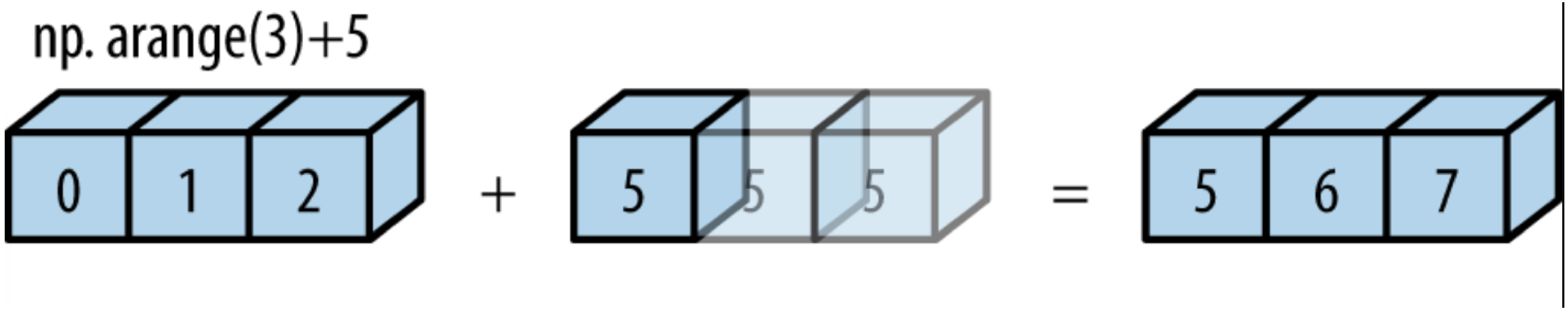
```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> b = np.ones((3, 4))
>>> b
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> a + b
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

The smaller array, **a** is being *stretched* or *broadcast* over the larger array, **b**'s second dimension to match its shape.

Source: *Python Data Science Handbook*, Jake VanderPlas

# Operations on NumPy Arrays

## Broadcasting

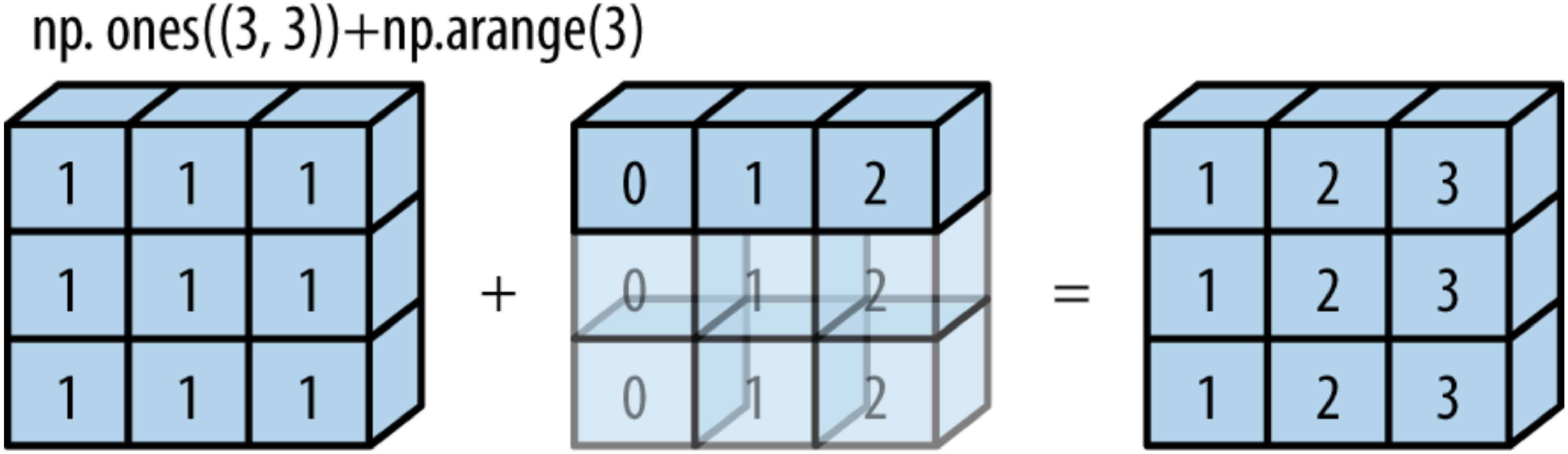



Note: No extra memory is allocated.

Source: *Python Data Science Handbook*, Jake VanderPlas

# Operations on NumPy Arrays

## Broadcasting



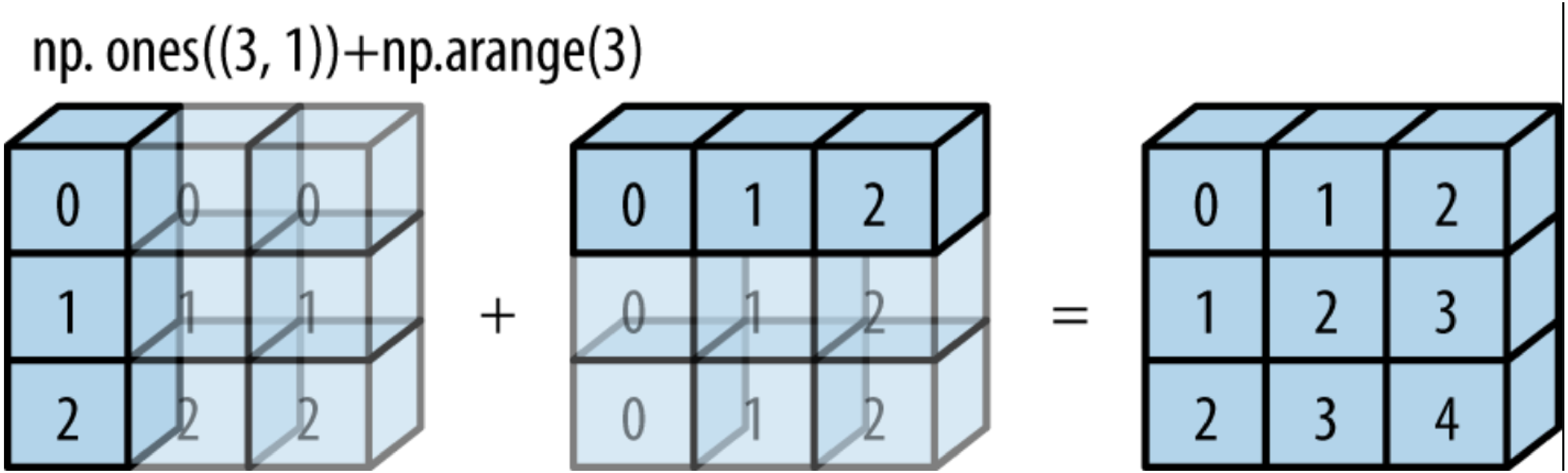

Note: No extra memory is allocated.

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Operations on NumPy Arrays

### Broadcasting




Note: No extra memory is allocated.

# Rules of Broadcasting

**Rule 1:**

If two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is *padded* with ones on its leading (left) side.

**Rule 2:**

If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.

**Rule 3:**

If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

Source: *Python Data Science Handbook*, Jake VanderPlas

# Comparison Operators as UFuncs

```
>>> a = np.arange(5)
>>> a > 3
array([False, False, False, False,  True], dtype=bool)
>>> a <= 4
array([ True,  True,  True,  True,  True], dtype=bool)
>>> a != 3
array([ True,  True,  True, False,  True], dtype=bool)
>>> (2 * a) == (a ** 2)
array([ True, False,  True, False, False], dtype=bool)
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Comparison Operators as UFuncs

| **Operator** | **UFunc** |
|---|---|
| == | np.equal |
| != | np.not_equal |
| < | np.less |
| <= | np.less_equal |
| > | np.greater |
| >= | np.greater_equal |

Source: *Python Data Science Handbook*, Jake VanderPlas

# Any and All

```
>>> r = np.random.randint(10, size=(3, 4))
>>> r
array([[3, 2, 9, 3],
       [5, 8, 1, 0],
       [1, 4, 1, 4]])
>>> r < 6
array([[ True,  True, False,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> np.any(r < 6)
True
>>> np.all(r < 9)
False
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Boolean Operators

With Python's bitwise logic operators (**&**, **|**, **^**, **~**), we can create more complex Boolean expressions.

```
>>> a = np.random.randint(100, size=(5,))
>>> a
array([66, 16, 23, 99, 22])
>>> # count values between 20 and 70, exclusive
>>> np.sum((a > 20) & (a < 70))
3
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Boolean Logical Operators as UFuncs

| Operator | UFunc |
|---|---|
| & | np.bitwise_and |
| \| | np.bitwise_or |
| ^ | np.bitwise_xor |
| ~ | np.bitwise_not |

We don't use the **and** and **or** keywords because they will effectively evaluate the truth or falsehood of the *entire object*. Instead, we're interested in the *bits within each object*. We're performing *multiple* Boolean evaluations on the content of the object.

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Boolean Arrays as Masks

Using Boolean arrays as masks to select particular subsets of an array can be a useful pattern.

```
>>> a = np.random.randint(100, size=(5,))
>>> a
array([66, 16, 23, 99, 22])
>>> # boolean array
>>> (a > 20) & (a < 70)
array([ True, False,  True, False,  True], dtype=bool)
>>> # mask original array with boolean array
>>> a[(a > 20) & (a < 70)]
array([66, 23, 22])
```

# Fancy Indexing

With fancy indexing, we can pass an array or list into square brackets to sample the original array. The shape of the result reflects the shape of the index arrays, rather than the shape of the array being indexed.

```
>>> a = np.random.randint(100, size=(10,))
>>> a
array([48, 93, 12, 88, 44, 31, 40, 32, 19,  1])
>>> indexes = [1, 5, 9]
>>> a[indexes]
array([93, 31,  1])
>>> indexes2 = np.array([[0, 9], [2, 7]])
>>> a[indexes2]
array([[48,  1],
       [12, 32]])
```

## Sorting Arrays

Python has built-in **sort** and **sorted** functions. NumPy's **np.sort** function is much more efficient. Uses O(n*lg n) quicksort, but mergesort and heapsort are available.

```
>>> a = np.random.randint(100, size=(10,))
>>> a
array([72, 43, 60, 21, 58, 21, 78, 33, 28, 29])
>>> np.sort(a)
array([21, 21, 28, 29, 33, 43, 58, 60, 72, 78])
>>> np.argsort(a)
array([3, 5, 8, 9, 7, 1, 4, 2, 0, 6])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Sorting Arrays Along Rows and Columns

Python has built-in **sort** and **sorted** functions. NumPy's **np.sort** function is much more efficient. Uses O(n*lg n) quicksort, but mergesort and heapsort are available.

```
>>> a = np.random.randint(100, size=(3, 10))
>>> a
array([[19, 41, 92, 85, 28, 44, 85, 95, 93,  0],
       [74, 25, 29, 29, 68, 80, 14, 54, 59, 36],
       [81, 30, 69, 18, 60, 26, 80, 30, 53, 49]])
>>> np.sort(a, axis=0)
array([[19, 25, 29, 18, 28, 26, 14, 30, 53,  0],
       [74, 30, 69, 29, 60, 44, 80, 54, 59, 36],
       [81, 41, 92, 85, 68, 80, 85, 95, 93, 49]])
>>> np.sort(a, axis=1)
array([[ 0, 19, 28, 41, 44, 85, 85, 92, 93, 95],
       [14, 25, 29, 29, 36, 54, 59, 68, 74, 80],
       [18, 26, 30, 30, 49, 53, 60, 69, 80, 81]])
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

# Structured Arrays

NumPy's *structured arrays* or *record arrays* provide efficient storage for compound, heterogeneous data.

```
>>> name = ['Rizzo', 'Schwarber', 'Bryant', 'Contreras']
>>> homeruns = [32, 30, 29, 21]
>>> age = [27, 24, 25, 25]
>>> # create structured array with all zeros
>>> x = np.zeros(4, dtype={'names':('name', 'homeruns', 'age'), 'formats':('U10', 'i4', 'i4
>>> x
array([('', 0, 0), ('', 0, 0), ('', 0, 0), ('', 0, 0)],
      dtype=[('name', '<U10'), ('homeruns', '<i4'), ('age', '<i4')])
>>> # fill array with lists of values
>>> x['name'] = name
>>> x['homeruns'] = homeruns
>>> x['age'] = age
>>> x
array([('Rizzo', 32, 27), ('Schwarber', 30, 24), ('Bryant', 29, 25),
       ('Contreras', 21, 25)],
      dtype=[('name', '<U10'), ('homeruns', '<i4'), ('age', '<i4')])
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Structured Arrays

NumPy's *structured arrays* or *record arrays* provide efficient storage for compound, heterogeneous data.

```
>>> name = ['Rizzo', 'Schwarber', 'Bryant', 'Contreras']
>>> homeruns = [32, 30, 29, 21]
>>> age = [27, 24, 25, 25]
>>> x = np.zeros(4, dtype={'names':('name', 'homeruns', 'age'), 'formats':('U10', 'i4', 'i4
>>> x['name'] = name
>>> x['homeruns'] = homeruns
>>> x['age'] = age
>>> x[0]
('Rizzo', 32, 27)
>>> x[0]['age']
27
>>> x[1]
'Schwarber'
>>> x[1]['homeruns']
30
```

# Python Scientific Stack

# Pandas

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas

Pandas builds on the structured data tools available in NumPy by giving us a data structure called a **DataFrame**, which acts as a multidimensional array with row and column labels, heterogeneous types, and/or missing data.

"As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs."

```
import pandas as pd
```

# Pandas: Series Object

A **Series** is a one-dimensional array of indexed data. It wraps:

- A sequence of values (accessible via **values** attribute).
- A sequence of indices (accessible via **index** attribute).

```
>>> import pandas as pd
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data
0     0.25
1     0.50
2     0.75
3     1.00
dtype: float64
>>> data.values
array([ 0.25,  0.5 ,  0.75,  1.  ])
>>> data.index
RangeIndex(start=0, stop=4, step=1)
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: Series Object

Data is accessible by offset (index) in square brackets.

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data
0     0.25
1     0.50
2     0.75
3     1.00
dtype: float64
>>> data[1]
0.5
>>> data[1:3]
1     0.50
2     0.75
dtype: float64
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: Series Object

We may consider a Pandas **Series** object as a generalized NumPy array. Whereas a NumPy array has an implicit integer index, a Pandas **Series** has an explicit index that may consist of values of any type.

```
>>> data = pd.Series(np.linspace(0.25, 1.0, 4),
... index=['a', 'b', 'c', 'd'])
>>> data
a     0.25
b     0.50
c     0.75
d     1.00
dtype: float64
>>> data['c']
0.75
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: Series Object

There exists no requirement that an index be sequential.

```
>>> data = pd.Series(np.linspace(0.25, 1, 4),
... index=[2, 5, 3, 7])
>>> data
2     0.25
5     0.50
3     0.75
7     1.00
dtype: float64
>>> data[5]
0.5
```

# Pandas: Series Object

We may also consider a Pandas **Series** a specialized dictionary. Whereas a Python **dict** maps a set of arbitrary keys to a set of arbitrary values, a **Series** maps a set of *typed* keys to a set of *typed* values.

"This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type of information of a Pandas **Series** makes it more efficient than a Python dictionary for certain operations."

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: Series Object

```
>>> cubs_hr = {'Rizzo': 32, 'Schwarber': 30,
...             'Bryant': 29, 'Contreras': 21}
>>> cubs_hr_series = pd.Series(cubs_hr)
>>> # note that the index becomes the sorted keys
>>> cubs_hr_series
Bryant       29
Contreras    21
Rizzo        32
Schwarber    30
dtype: int64
>>> cubs_hr_series['Rizzo']
32
```

# Pandas: Series Object

The **Series** supports array-style operations, like slicing:

```
>>> cubs_hr = {'Rizzo': 32, 'Schwarber': 30,
...            'Bryant': 29, 'Contreras': 21}
>>> cubs_hr_series = pd.Series(cubs_hr)
>>> cubs_hr_series['Bryant':'Rizzo']
Bryant       29
Contreras    21
Rizzo        32
dtype: int64
>>> cubs_hr_series.index
Index(['Bryant', 'Contreras', 'Rizzo', 'Schwarber'], dtype='object')
>>> cubs_hr_series.values
array([29, 21, 32, 30])
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: Series Object

Pandas **Series** can be created from

- Lists, NumPy arrays: **index** defaults to sequence of integers.
- Dictionaries: **index** defaults to sorted keys of the dictionary.
- Scalars: value repeated to fill given **index**.

```
>>> pd.Series(5, index=[100, 200, 300])
100    5
200    5
300    5
dtype: int64
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: DataFrame Object

"If a **Series** is an analog of a one-dimensional array with flexible indices, a **DataFrame** is an analog of a two-dimensional array with both flexible row indices and flexible column names."

"Just as you might think of a two-dimensional array as an ordered sequence of aligned one-dimensional columns, you can think of a **DataFrame** as a sequence of aligned **Series** objects. Here, by 'aligned' we mean that they share the same index."

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: DataFrame Object

```
>>> hr_dict = {'Rizzo': 32, 'Schwarber': 30,
...             'Bryant': 29, 'Contreras': 21}
>>> avg_dict = {'Rizzo': .273, 'Schwarber': .211,
...             'Bryant': .295, 'Contreras': .276}
>>> hr = pd.Series(hr_dict)
>>> avg = pd.Series(avg_dict)
>>> cubs = pd.DataFrame({'home_runs': hr, 'batting_average': avg})
>>> cubs
           batting_average  home_runs
Bryant               0.295         29
Contreras            0.276         21
Rizzo                0.273         32
Schwarber            0.211         30
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: DataFrame Object

A **DataFrame** has attributes:

- **index**: An **Index** object. The values are the row/index labels.
- **columns**: An **index** object. The values are the column labels.

```
>>> cubs
           batting_average  home_runs
Bryant               0.295         29
Contreras            0.276         21
Rizzo                0.273         32
Schwarber            0.211         30
>>> cubs.index
Index(['Bryant', 'Contreras', 'Rizzo', 'Schwarber'], dtype='object')
>>> cubs.columns
Index(['batting_average', 'home_runs'], dtype='object')
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: DataFrame Object

Another way to frame our understanding of the **DataFrame** object is to consider it a specialized dictionary. Whereas a dictionary maps arbitrary keys to arbitrary values, a **DataFrame** maps a column name to a **Series** of column data.

```
>>> cubs
            batting_average  home_runs
Bryant                0.295         29
Contreras             0.276         21
Rizzo                 0.273         32
Schwarber             0.211         30
>>> cubs['batting_average']
Bryant       0.295
Contreras    0.276
Rizzo        0.273
Schwarber    0.211
Name: batting_average, dtype: float64
```

```
>>> cubs['home_runs']
Bryant       29
Contreras    21
Rizzo        32
Schwarber    30
Name: home_runs, dtype: int64
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Pandas: DataFrame Object

Because the __getitem__ behavior of a **DataFrame** returns a column, our conceptualization of the **DataFrame** as a two-dimensional ndarray may be misleading. For this reason, the specialized dictionary conceptualization is preferable.

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Indexing and Selection

```
>>> data = pd.Series(np.linspace(.25, 1, 4),
...                  index=['a', 'b', 'c', 'd'])
>>> data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
>>> # access element by index like a dictionary
>>> data['b']
0.5
>>> # access element by implicit integer index
>>> data[2]
0.75
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Indexing and Selection: Series

```
>>> # extend series
>>> data['e'] = 1.25
>>> # slicing by explicit index
>>> data['a':'c']
a     0.25
b     0.50
c     0.75
dtype: float64
>>> # slicing by implicit index
>>> data[0:2]
a     0.25
b     0.50
dtype: float64
```

```
>>> # masking
>>> data[(data > 0.3) & (data < 0.8)]
b     0.50
c     0.75
dtype: float64
>>> # fancy indexing
>>> data[['a', 'e']]
a     0.25
e     1.25
dtype: float64
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Indexers: Series

```
>>> data = pd.Series(np.linspace(.25, 1, 4), index=['a', 'b', 'c', 'd'])
>>> # always references explicit index
>>> data.loc['a']
0.25
>>> data.loc['a':'c']
a     0.25
b     0.50
c     0.75
dtype: float64
>>> # always references implicit index
>>> data.iloc[1]
0.5
>>> data.iloc[1:3]
b     0.50
c     0.75
dtype: float64
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

## Data Selection: DataFrames

```
>>> ab = pd.Series({'Rizzo': 572, 'Bryant': 549, 'Baez': 469, 'Zobrist': 435})
>>> hits = pd.Series({'Rizzo': 156, 'Bryant': 162, 'Baez': 128, 'Zobrist': 101})
>>> cubs = pd.DataFrame({'at_bats': ab, 'hits': hits})
>>> cubs['hits']  # dictionary-style indexing
Baez        128
Bryant      162
Rizzo       156
Zobrist     101
Name: hits, dtype: int64
>>> cubs.hits  # attribute-style access with column names that are strings
Baez        128
Bryant      162
Rizzo       156
Zobrist     101
Name: hits, dtype: int64
>>> cubs['hits'] is cubs.hits
True
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Selection: DataFrames

```
>>> ab = pd.Series({'Rizzo': 572, 'Bryant': 549, 'Baez': 469, 'Zobrist': 435})
>>> hits = pd.Series({'Rizzo': 156, 'Bryant': 162, 'Baez': 128, 'Zobrist': 101})
>>> cubs = pd.DataFrame({'at_bats': ab, 'hits': hits})
>>> cubs
         at_bats  hits
Baez         469   128
Bryant       549   162
Rizzo        572   156
Zobrist      435   101
>>> # add a new column using dictionary-style assignment
>>> cubs['avg'] = cubs['hits'] / cubs['at_bats']
>>> cubs
         at_bats  hits       avg
Baez         469   128  0.272921
Bryant       549   162  0.295082
Rizzo        572   156  0.272727
Zobrist      435   101  0.232184
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Python Programming

## Data Selection: DataFrames

```
>>> cubs
         at_bats  hits       avg
Baez         469   128  0.272921
Bryant       549   162  0.295082
Rizzo        572   156  0.272727
Zobrist      435   101  0.232184
>>> # transpose
>>> cubs.T
               Baez      Bryant       Rizzo     Zobrist
at_bats  469.000000  549.000000  572.000000  435.000000
hits     128.000000  162.000000  156.000000  101.000000
avg        0.272921    0.295082    0.272727    0.232184
>>> # values as 2-d array
>>> cubs.values
array([[  4.69000000e+02,   1.28000000e+02,   2.72921109e-01],
       [  5.49000000e+02,   1.62000000e+02,   2.95081967e-01],
       [  5.72000000e+02,   1.56000000e+02,   2.72727273e-01],
       [  4.35000000e+02,   1.01000000e+02,   2.32183908e-01]])
```

# Python Programming

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Indexing: DataFrames

```
>>> cubs.loc['Rizzo']
at_bats     572.000000
hits        156.000000
avg           0.272727
Name: Rizzo, dtype: float64
>>> cubs.loc['Rizzo', 'hits']
156
>>> cubs.iloc[0]
at_bats     469.000000
hits        128.000000
avg           0.272921
Name: Baez, dtype: float64
>>> cubs.iloc[0, 0]
469
```

```
>>> cubs.loc['Rizzo':'Zobrist', 'hits']
Rizzo      156
Zobrist    101
Name: hits, dtype: int64
>>> cubs.loc['Rizzo':'Zobrist']
         at_bats  hits       avg
Rizzo        572   156  0.272727
Zobrist      435   101  0.232184
>>> cubs.iloc[0:2]
        at_bats  hits       avg
Baez        469   128  0.272921
Bryant      549   162  0.295082
>>> cubs.iloc[0:2, 1:2]
        hits
Baez     128
Bryant   162
```

Source: *Python Data Science Handbook*, Jake VanderPlas

# Data Indexing: DataFrames

```
>>> # masking
>>> cubs[cubs['at_bats'] < 500]
         at_bats  hits       avg
Baez         469   128  0.272921
Zobrist      435   101  0.232184
>>> cubs[(cubs['at_bats'] < 500) & (cubs['hits'] > 120)]
      at_bats  hits       avg
Baez      469   128  0.272921
```

# DataFrame Operations

Pandas **DataFrame** objects inherit efficient element-wise operations from NumPy. Additionally, **DataFrame** objects "include a couple of useful twists":

For unary operations, ... ufuncs will *preserve index and column labels* in the output.

```
>>> df = pd.DataFrame(np.random.randint(0, 10, (3, 4)),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B  C  D
0  8  3  6  9
1  1  0  4  2
2  5  8  3  7
>>> np.sin(df * np.pi / 4)
              A             B             C         D
0 -2.449294e-16  7.071068e-01 -1.000000e+00  0.707107
1  7.071068e-01  0.000000e+00  1.224647e-16  1.000000
2 -7.071068e-01 -2.449294e-16  7.071068e-01 -0.707107
```

# Creation of DataFrame from File

```
>>> df_from_csv = pd.read_csv('/path/to/file.csv')
>>> df_from_web_csv = pd.read_csv('http://somewebresource.com/resource')
>>> df_from_json = pd.read_json('/path/to/file.json')
>>> df_from_web_json = pd.read_json('http://somewebresource.com/resource')
```

# Python Scientific Stack

# matplotlib

Source: *Python Data Science Handbook*, Jake VanderPlas

# Matplotlib

"Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for creating interactive MATLAB-style plotting via gnuplot from the IPython command line."

```
import matplotlib as mpl
import matplotlib.pyplot as plt

# we will focus on plt
```

# Sources

- *Learning Python*, Mark Lutz, O'Reilly
- *Data Science Handbook*, Jake VanderPlas, O'Reilly