7. Array-Oriented Programming with NumPy

Objectives

In this chapter, you'll:

- Learn what arrays are and how they differ from lists.
- Use the numpy module's highperformance ndarrays.
- Compare list and ndarray performance with the IPython %timeit magic.
- Use ndarrays to store and retrieve data efficiently.
- Create and initialize ndarrays.

Objectives (cont.)

- Refer to individual ndarray elements.
- Iterate through ndarrays.
- Create and manipulate multidimensional ndarrays.
- Perform common ndarray manipulations.
- Create and manipulate pandas one-dimensional Series and two-dimensional DataFrames.

Objectives (cont.)

- Customize Series and DataFrame indices.
- Calculate basic descriptive statistics for data in a Series and a DataFrame.
- Customize floating-point number precision in pandas output formatting.

Outline

Outline (cont.)

Outline (cont.)

7.1 Introduction

NumPy (Numerical Python) Library

- First appeared in 2006 and is the **preferred Python array implementation**.
- High-performance, richly functional ***n*-dimensional array** type called **`ndarray`**.
- **Written in C** and **up to 100 times faster than lists**.
- Critical in big-data processing, AI applications and much more.
- According to `libraries.io`, **over 450 Python libraries depend on NumPy**.
- Many popular data science libraries such as Pandas, SciPy (Scientific Python) and Keras (for deep learning) are built on or depend on NumPy.

Array-Oriented Programming

- **Functional-style programming** with **internal iteration** makes array-oriented manipulations concise and straightforward, and reduces the possibility of error.

7.2 Creating `array`s from Existing Data

- Creating an array with the **`array`** function
- Argument is an `array` or other iterable
- Returns a new `array` containing the argument’s elements

In [1]:

```
import numpy as np
```

In [2]:

```
numbers = np.array([2, 3, 5, 7, 11])
```

In [3]:

```
type(numbers)
```

Out[3]:

```
numpy.ndarray
```

In [4]:

```
numbers
```

Out[4]:

```
array([ 2,  3,  5,  7, 11])
```

Multidimensional Arguments

In [5]:

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

Out[5]:

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

7.3 `array` Attributes

- **attributes** enable you to discover information about its structure and contents

In [1]:

```
import numpy as np
```

In [2]:

```
integers = np.array([[1, 2, 3], [4, 5, 6]])
```

In [3]:

```
integers
```

Out[3]:

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

In [4]:

```
floats = np.array([0.0, 0.1, 0.2, 0.3, 0.4])
```

In [5]:

```
floats
```

Out[5]:

```
array([0. , 0.1, 0.2, 0.3, 0.4])
```

- NumPy does not display trailing 0s

Determining an `array`'s Element Type

In [6]:

```
integers.dtype
```

Out[6]:

```
dtype('int64')
```

In [7]:

```
floats.dtype
```

Out[7]:

```
dtype('float64')
```

- For performance reasons, NumPy is written in the C programming language and uses C's data types
- [Other NumPy types (https://docs.scipy.org/doc/numpy/user/basics.types.html)](https://docs.scipy.org/doc/numpy/user/basics.types.html)

Determining an `array`'s Dimensions

- **`ndim`** contains an `array`'s number of dimensions
- **`shape`** contains a *tuple* specifying an `array`'s dimensions

In [8]:

```
integers.ndim
```

Out[8]:

```
2
```

In [9]:

```
floats.ndim
```

Out[9]:

```
1
```

In [10]:

```
integers.shape
```

Out[10]:

```
(2, 3)
```

In [11]:

```
floats.shape
```

Out[11]:

```
(5,)
```

Determining an `array`'s Number of Elements and Element Size

- view an `array`'s total number of elements with **`size`**
- view number of bytes required to store each element with **`itemsize`**

In [12]:

```
integers.size
```

Out[12]:

```
6
```

In [13]:

```
integers.itemsize
```

Out[13]:

```
8
```

In [14]:

```
floats.size
```

Out[14]:

```
5
```

In [15]:

```
floats.itemsize
```

Out[15]:

```
8
```

Iterating through a Multidimensional `array`'s Elements

In [16]:

```
for row in integers:
    for column in row:
        print(column, end='  ')
    print()
```

```
1  2  3  
4  5  6  
```

- Iterate through a multidimensional `array` as if it were one-dimensional by using **`flat`**

In [17]:

```
for i in integers.flat:
    print(i, end='  ')
```

```
1  2  3  4  5  6  
```

7.4 Filling `array`s with Specific Values

- Functions **`zeros`**, **`ones`** and **`full`** create `array`s containing `0`s, `1`s or a specified value, respectively

In [1]:

```
import numpy as np
```

In [2]:

```
np.zeros(5)
```

Out[2]:

```
array([0., 0., 0., 0., 0.])
```

- For a tuple of integers, these functions return a multidimensional `array` with the specified dimensions

In [3]:

```
np.ones((2, 4), dtype=int)
```

Out[3]:

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

In [4]:

```
np.full((3, 5), 13)
```

Out[4]:

```
array([[13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13],
       [13, 13, 13, 13, 13]])
```

7.5 Creating `array`s from Ranges

- NumPy provides optimized functions for creating `array`s from ranges

Creating Integer Ranges with `arange`

In [1]:

```
import numpy as np
```

In [2]:

```
np.arange(5)
```

Out[2]:

```
array([0, 1, 2, 3, 4])
```

In [3]:

```
np.arange(5, 10)
```

Out[3]:

```
array([5, 6, 7, 8, 9])
```

In [4]:

```
np.arange(10, 1, -2)
```

Out[4]:

```
array([10,  8,  6,  4,  2])
```

Creating Floating-Point Ranges with `linspace`

- Produce evenly spaced floating-point ranges with NumPy's **`linspace`** function
- Ending value **is included** in the `array`

In [5]:

```
np.linspace(0.0, 1.0, num=5)
```

Out[5]:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Reshaping an `array`

- `array` method **`reshape`** transforms an array into different number of dimensions
- New shape must have the **same** number of elements as the original

In [6]:

```
np.arange(1, 21).reshape(4, 5)
```

Out[6]:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

Displaying Large `array` s

- When displaying an `array` , if there are 1000 items or more, NumPy drops the middle rows, columns or both from the output

In [7]:

```
np.arange(1, 100001).reshape(4, 25000)
```

Out[7]:

```
array([[     1,      2,      3, ...,  24998,  24999,  25000],
       [ 25001,  25002,  25003, ...,  49998,  49999,  50000],
       [ 50001,  50002,  50003, ...,  74998,  74999,  75000],
       [ 75001,  75002,  75003, ...,  99998,  99999, 100000]])
```

In [8]:

```
np.arange(1, 100001).reshape(100, 1000)
```

Out[8]:

```
array([[     1,      2,      3, ...,    998,    999,   1000],
       [  1001,   1002,   1003, ...,   1998,   1999,   2000],
       [  2001,   2002,   2003, ...,   2998,   2999,   3000],
       ...,
       [ 97001,  97002,  97003, ...,  97998,  97999,  98000],
       [ 98001,  98002,  98003, ...,  98998,  98999,  99000],
       [ 99001,  99002,  99003, ...,  99998,  99999, 100000]])
```

Arithmetic Operations with `array`s and Individual Numeric Values

7.7 `array` Operators

- `array` operators perform operations on **entire `array`s.**
- Can perform arithmetic **between `array`s and scalar numeric values**, and **between `array`s of the same shape**.

In [1]:

```
import numpy as np
```

In [2]:

```
numbers = np.arange(1, 6)
```

In [3]:

```
numbers
```

Out[3]:

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

In [4]:

```
numbers * 2
```

Out[4]:

```
array([ 2,  4,  6,  8, 10])
```

In [5]:

```
numbers ** 3
```

Out[5]:

```
array([  1,   8,  27,  64, 125])
```

In [6]:

```
numbers  # numbers is unchanged by the arithmetic operators
```

Out[6]:

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

In [7]:

```
numbers += 10
```

In [8]:

```
numbers
```

Out[8]:

```
array([11, 12, 13, 14, 15])
```

Broadcasting

- Arithmetic operations require as operands two `array` s of the **same size and shape**.
- **`numbers * 2`** is equivalent to **`numbers * [2, 2, 2, 2, 2]`** for a 5-element array.
- Applying the operation to every element is called **broadcasting**.
- Also can be applied between `array` s of different sizes and shapes, enabling some concise and powerful manipulations.

Arithmetic Operations Between `array` s

- Can perform arithmetic operations and augmented assignments between `array` s of the *same* shape

In [9]:

```
numbers2 = np.linspace(1.1, 5.5, 5)
```

In [10]:

```
numbers2
```

Out[10]:

```
array([1.1, 2.2, 3.3, 4.4, 5.5])
```

In [11]:

```
numbers * numbers2
```

Out[11]:

```
array([12.1, 26.4, 42.9, 61.6, 82.5])
```

Comparing `array` s

- Can compare `array` s with individual values and with other `array` s
- Comparisons performed **element-wise**
- Produce `array` s of Boolean values in which each element's `True` or `False` value indicates the comparison result

In [12]:

```
numbers
```

Out[12]:

```
array([11, 12, 13, 14, 15])
```

In [13]:

```
numbers >= 13
```

Out[13]:

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

In [14]:

```
numbers2
```

Out[14]:

```
array([1.1, 2.2, 3.3, 4.4, 5.5])
```

In [15]:

```
numbers2 < numbers
```

Out[15]:

```
array([ True,  True,  True,  True,  True])
```

In [16]:

```
numbers == numbers2
```

Out[16]:

```
array([False, False, False, False, False])
```

In [17]:

```
numbers == numbers
```

Out[17]:

```
array([ True,  True,  True,  True,  True])
```

7.8 NumPy Calculation Methods

- These methods **ignore the `array`'s shape** and **use all the elements in the calculations**.
- Consider an `array` representing four students' grades on three exams:

In [1]:

```
import numpy as np
```

In [2]:

```
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
```

In [3]:

```
grades
```

Out[3]:

```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

- Can use methods to calculate **`sum`**, **`min`**, **`max`**, **`mean`**, **`std`** (standard deviation) and **`var`** (variance)
- Each is a functional-style programming **reduction**

In [4]:

```
grades.sum()
```

Out[4]:

```
1054
```

In [5]:

```
grades.min()
```

Out[5]:

```
70
```

In [6]:

```
grades.max()
```

Out[6]:

```
100
```

In [7]:

```
grades.mean()
```

Out[7]:

```
87.83333333333333
```

In [8]:

```
grades.std()
```

Out[8]:

```
8.792357792739987
```

In [9]:

```
grades.var()
```

Out[9]:

```
77.30555555555556
```

Calculations by Row or Column

- You can perform calculations by column or row (or other dimensions in arrays with more than two dimensions)
- Each 2D+ array has **one axis per dimension** (https://docs.scipy.org/doc/numpy-1.16.0/glossary.html)
- In a 2D array, `axis=0` indicates calculations should be **column-by-column**

In [10]:

```
grades.mean(axis=0)
```

Out[10]:

```
array([95.25, 85.25, 83.  ])
```

- In a 2D array, `axis=1` indicates calculations should be **row-by-row**

In [11]:

```
grades.mean(axis=1)
```

Out[11]:

```
array([84.33333333, 92.33333333, 87.        , 87.66666667])
```

- Other Numpy `array` Calculation Methods (https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html)

7.9 Universal Functions

- Standalone **[universal functions (ufuncs)](https://docs.scipy.org/doc/numpy/reference/ufuncs.html)** (https://docs.scipy.org/doc/numpy/reference/ufuncs.html) perform **element-wise operations** using one or two `array` or array-like arguments (like lists)
- Each returns a **new `array`** containing the results
- Some ufuncs are called when you use `array` operators like `+` and `*`

- Create an `array` and calculate the square root of its values, using the **`sqrt` universal function**

In [1]:

```
import numpy as np
```

In [2]:

```
numbers = np.array([1, 4, 9, 16, 25, 36])
```

In [3]:

```
np.sqrt(numbers)
```

Out[3]:

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

- Add two `array` s with the same shape, using the **`add` universal function**
- Equivalent to:

```
numbers + numbers2
```

In [4]:

```
numbers2 = np.arange(1, 7) * 10
```

In [5]:

```
numbers2
```

Out[5]:

```
array([10, 20, 30, 40, 50, 60])
```

In [6]:

```
np.add(numbers, numbers2)
```

Out[6]:

```
array([11, 24, 39, 56, 75, 96])
```

Broadcasting with Universal Functions

- Universal functions can use broadcasting, just like NumPy `array` operators

In [7]:

```
np.multiply(numbers2, 5)
```

Out[7]:

```
array([ 50, 100, 150, 200, 250, 300])
```

In [8]:

```
numbers3 = numbers2.reshape(2, 3)
```

In [9]:

```
numbers3
```

Out[9]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [10]:

```
numbers4 = np.array([2, 4, 6])
```

In [11]:

```
np.multiply(numbers3, numbers4)
```

Out[11]:

```
array([[ 20,  80, 180],
       [ 80, 200, 360]])
```

- [Broadcasting rules documentation (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)](https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)

Other Universal Functions

NumPy universal functions
Math — `add` , `subtract` , `multiply` , `divide` , `remainder` , `exp` , `log` , `sqrt` , `power` , and more.
Trigonometry — `sin` , `cos` , `tan` , `hypot` , `arcsin` , `arccos` , `arctan` , and more.
Bit manipulation — `bitwise_and` , `bitwise_or` , `bitwise_xor` , `invert` , `left_shift` and `right_shift` .
Comparison — `greater` , `greater_equal` , `less` , `less_equal` , `equal` , `not_equal` , `logical_and` , `logical_or` , `logical_xor` , `logical_not` , `minimum` , `maximum` , and more.
Floating point — `floor` , `ceil` , `isinf` , `isnan` , `fabs` , `trunc` , and more.

7.10 Indexing and Slicing

- One-dimensional `array` s can be **indexed** and **sliced** like lists.

Indexing with Two-Dimensional `array` s

- To select an element in a two-dimensional `array` , specify a tuple containing the element's row and column indices in square brackets

In [1]:

```
import numpy as np
```

In [2]:

```
grades = np.array([[87, 96, 70], [100, 87, 90],
                   [94, 77, 90], [100, 81, 82]])
```

In [3]:

```
grades
```

Out[3]:

```
array([[ 87,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

In [4]:

```
grades[0, 1]  # row 0, column 1
```

Out[4]:

```
96
```

Selecting a Subset of a Two-Dimensional `array` 's Rows

- To select a single row, specify only one index in square brackets

In [5]:

```
grades[1]
```

Out[5]:

```
array([100,  87,  90])
```

- Select multiple sequential rows with slice notation

In [6]:

```
grades[0:2]
```

Out[6]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

- Select multiple non-sequential rows with a list of row indices

In [7]:

```
grades[[1, 3]]
```

Out[7]:

```
array([[100,  87,  90],
       [100,  81,  82]])
```

Selecting a Subset of a Two-Dimensional `array`'s Columns

- The **column index** also can be a specific **index**, a **slice** or a **list**

In [8]:

```
grades[:, 0]
```

Out[8]:

```
array([ 87, 100,  94, 100])
```

In [9]:

```
grades[:, 1:3]
```

Out[9]:

```
array([[96, 70],
       [87, 90],
       [77, 90],
       [81, 82]])
```

In [10]:

```
grades[:, [0, 2]]
```

Out[10]:

```
array([[ 87,  70],
       [100,  90],
       [ 94,  90],
       [100,  82]])
```

7.11 Views: Shallow Copies

- Views “see” the data in other objects, rather than having their own copies of the data
- Views are shallow copies * `array` method **`view`** returns a **new** array object with a **view** of the original `array` object’s data

In [1]:

```
import numpy as np
```

In [2]:

```
numbers = np.arange(1, 6)
```

In [3]:

```
numbers
```

Out[3]:

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

In [4]:

```
numbers2 = numbers.view()
```

In [5]:

```
numbers2
```

Out[5]:

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

- Use built-in `id` function to see that `numbers` and `numbers2` are **different** objects

In [6]:

```
id(numbers)
```

Out[6]:

```
4431803056
```

In [7]:

```
id(numbers2)
```

Out[7]:

```
4430398928
```

- Modifying an element in the original `array` , also modifies the view and vice versa

In [8]:

```
numbers[1] *= 10
```

In [9]:

```
numbers2
```

Out[9]:

```
array([ 1, 20,  3,  4,  5])
```

In [10]:

```
numbers
```

Out[10]:

```
array([ 1, 20,  3,  4,  5])
```

In [11]:

```
numbers2[1] /= 10
```

In [12]:

```
numbers
```

Out[12]:

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

In [13]:

```
numbers2
```

Out[13]:

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

Slice Views

- Slices also create views

In [14]:

```
numbers2 = numbers[0:3]
```

In [15]:

```
numbers2
```

Out[15]:

```
array([1, 2, 3])
```

In [16]:

```
id(numbers)
```

Out[16]:

```
4431803056
```

In [17]:

```
id(numbers2)
```

Out[17]:

```
4451350368
```

- Confirm that `numbers2` is a view of only first three `numbers` elements

In [18]:

```
numbers2[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call l
ast)
<ipython-input-18-83bd44fddddf> in <module>
----> 1 numbers2[3]

IndexError: index 3 is out of bounds for axis 0 with size 3
```

- Modify an element both `array` s share to show both are updated

In [19]:

```
numbers[1] *= 20
```

In [20]:

```
numbers
```

Out[20]:

```
array([ 1, 40,  3,  4,  5])
```

In [21]:

```
numbers2
```

Out[21]:

```
array([ 1, 40,  3])
```

7.12 Deep Copies

* When sharing **mutable** values, sometimes it’s necessary to create a **deep copy** of the original data
* Especially important in multi-core programming, where separate parts of your program could attempt to modify your data at the same time, possibly corrupting it

* **`array` method `copy`** returns a new array object with an independent copy of the original array's data

In [1]:

```
import numpy as np
```

In [2]:

```
numbers = np.arange(1, 6)
```

In [3]:

```
numbers
```

Out[3]:

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

In [4]:

```
numbers2 = numbers.copy()
```

In [5]:

```
numbers2
```

Out[5]:

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

In [6]:

```
numbers[1] *= 10
```

In [7]:

```
numbers
```

Out[7]:

```
array([ 1, 20,  3,  4,  5])
```

In [8]:

```
numbers2
```

Out[8]:

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

Module `copy` —Shallow vs. Deep Copies for Other Types of Python Objects

---

7.13 Reshaping and Transposing

`reshape` vs. `resize`

- Method `reshape` returns a *view* (shallow copy) of the original `array` with new dimensions
- Does *not* modify the original `array`

In [1]:

```
import numpy as np
```

In [2]:

```
grades = np.array([[87, 96, 70], [100, 87, 90]])
```

In [3]:

```
grades
```

Out[3]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In [4]:

```
grades.reshape(1, 6)
```

Out[4]:

```
array([[ 87,  96,  70, 100,  87,  90]])
```

In [5]:

```
grades
```

Out[5]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

- Method `resize` modifies the original `array`'s shape

In [6]:

```
grades.resize(1, 6)
```

In [7]:

```
grades
```

Out[7]:

```
array([[ 87,  96,  70, 100,  87,  90]])
```

`flatten` vs. `ravel`

- Can flatten a multi-dimensonal array into a single dimension with methods **`flatten`** and **`ravel`**
- `flatten` *deep copies* the original array's data

In [8]:

```
grades = np.array([[87, 96, 70], [100, 87, 90]])
```

In [9]:

```
grades
```

Out[9]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In [10]:

```
flattened = grades.flatten()
```

In [11]:

```
flattened
```

Out[11]:

```
array([ 87,  96,  70, 100,  87,  90])
```

In [12]:

```
grades
```

Out[12]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In [13]:

```
flattened[0] = 100
```

In [14]:

```
flattened
```

Out[14]:

```
array([100,  96,  70, 100,  87,  90])
```

In [15]:

```
grades
```

Out[15]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

- Method `ravel` produces a *view* of the original `array`, which *shares* the `grades` `array`'s data

In [16]:

```
raveled = grades.ravel()
```

In [17]:

```
raveled
```

Out[17]:

```
array([ 87,  96,  70, 100,  87,  90])
```

In [18]:

```
grades
```

Out[18]:

```
array([[ 87,  96,  70],
       [100,  87,  90]])
```

In [19]:

```
raveled[0] = 100
```

In [20]:

```
raveled
```

Out[20]:

```
array([100,  96,  70, 100,  87,  90])
```

In [21]:

```
grades
```

Out[21]:

```
array([[100,  96,  70],
       [100,  87,  90]])
```

Transposing Rows and Columns

- Can quickly **transpose** an `array`'s rows and columns
  - "flips" the `array`, so the rows become the columns and the columns become the rows
- **`T` attribute** returns a transposed *view* (shallow copy) of the `array`

In [22]:

```
grades.T
```

Out[22]:

```
array([[100, 100],
       [ 96,  87],
       [ 70,  90]])
```

In [23]:

```
grades
```

Out[23]:

```
array([[100,  96,  70],
       [100,  87,  90]])
```

Horizontal and Vertical Stacking

- Can combine arrays by adding more columns or more rows—known as *horizontal stacking* and *vertical stacking*

In [24]:

```
grades2 = np.array([[94, 77, 90], [100, 81, 82]])
```

- Combine `grades` and `grades2` with NumPy's **`hstack`** **(horizontal stack) function** by passing a tuple containing the arrays to combine
- The extra parentheses are required because `hstack` expects one argument
- Adds more columns

In [25]:

```
np.hstack((grades, grades2))
```

Out[25]:

```
array([[100,  96,  70,  94,  77,  90],
       [100,  87,  90, 100,  81,  82]])
```

- Combine `grades` and `grades2` with NumPy's **`vstack`** **(vertical stack) function**
- Adds more rows

In [26]:

```
np.vstack((grades, grades2))
```

Out[26]:

```
array([[100,  96,  70],
       [100,  87,  90],
       [ 94,  77,  90],
       [100,  81,  82]])
```

7.14.1 pandas `Series`

- An enhanced one-dimensional `array`
- Supports custom indexing, including even non-integer indices like strings
- Offers additional capabilities that make them more convenient for many data-science oriented tasks
  - `Series` may have missing data
  - Many `Series` operations ignore missing data by default

Creating a `Series` with Default Indices

- By default, a `Series` has integer indices numbered sequentially from 0

In [1]:

```
import pandas as pd
```

In [2]:

```
grades = pd.Series([87, 100, 94])
```

Creating a `Series` with All Elements Having the Same Value

- Second argument is a one-dimensional iterable object (such as a list, an `array` or a `range`) containing the `Series`' indices
- Number of indices determines the number of elements

In [149]:

```
pd.Series(98.6, range(3))
```

Out[149]:

```
0    98.6
1    98.6
2    98.6
dtype: float64
```

Accessing a `Series`' Elements

In [150]:

```
grades[0]
```

Out[150]:

```
87
```

Producing Descriptive Statistics for a Series

- `Series` provides many methods for common tasks including producing various descriptive statistics
- Each of these is a functional-style reduction

In [151]:

```
grades.count()
```

Out[151]:

```
3
```

In [152]:

```
grades.mean()
```

Out[152]:

```
93.66666666666667
```

In [153]:

```
grades.min()
```

Out[153]:

```
87
```

In [154]:

```
grades.max()
```

Out[154]:

```
100
```

In [155]:

```
grades.std()
```

Out[155]:

```
6.506407098647712
```

- `Series` method **`describe`** produces all these stats and more
- The `25%` , `50%` and `75%` are **quartiles**:
  - `50%` represents the median of the sorted values.
  - `25%` represents the median of the first half of the sorted values.
  - `75%` represents the median of the second half of the sorted values.
- For the quartiles, if there are two middle elements, then their average is that quartile's median

In [156]:

```
grades.describe()
```

Out[156]:

```
count       3.000000
mean       93.666667
std         6.506407
min        87.000000
25%        90.500000
50%        94.000000
75%        97.000000
max       100.000000
dtype: float64
```

Creating a `Series` with Custom Indices

Can specify custom indices with the `index` keyword argument

In [157]:

```
grades = pd.Series([87, 100, 94], index=['Wally', 'Eva', 'Sam'])
```

In [158]:

```
grades
```

Out[158]:

```
Wally     87
Eva      100
Sam       94
dtype: int64
```

Dictionary Initializers

- If you initialize a `Series` with a dictionary, its keys are the indices, and its values become the `Series`' element values

In [159]:

```
grades = pd.Series({'Wally': 87, 'Eva': 100, 'Sam': 94})
```

In [160]:

```
grades
```

Out[160]:

```
Wally     87
Eva      100
Sam       94
dtype: int64
```

Accessing Elements of a `Series` Via Custom Indices

- Can access individual elements via square brackets containing a custom index value

In [161]:

```
grades['Eva']
```

Out[161]:

```
100
```

- If custom indices are strings that could represent valid Python identifiers, pandas automatically adds them to the `Series` as attributes

In [162]:

```
grades.Wally
```

Out[162]:

```
87
```

- **`dtype` attribute** returns the underlying `array`'s element type

In [163]:

```
grades.dtype
```

Out[163]:

```
dtype('int64')
```

- **`values` attribute** returns the underlying `array`

In [164]:

```
grades.values
```

Out[164]:

```
array([ 87, 100,  94])
```

Creating a Series of Strings

- In a `Series` of strings, you can use **`str` attribute** to call string methods on the elements

In [165]:

```
hardware = pd.Series(['Hammer', 'Saw', 'Wrench'])
```

In [166]:

```
hardware
```

Out[166]:

```
0    Hammer
1       Saw
2    Wrench
dtype: object
```

- Call string method `contains` on each element
- Returns a `Series` containing `bool` values indicating the `contains` method's result for each element
- The `str` attribute provides many string-processing methods that are similar to those in Python's string type
  - https://pandas.pydata.org/pandas-docs/stable/api.html#string-handling (https://pandas.pydata.org/pandas-docs/stable/api.html#string-handling)

In [167]:

```
hardware.str.contains('a')
```

Out[167]:

```
0     True
1     True
2    False
dtype: bool
```

- Use string method `upper` to produce a *new* `Series` containing the uppercase versions of each element in `hardware`

In [168]:

```
hardware.str.upper()
```

Out[168]:

```
0    HAMMER
1       SAW
2    WRENCH
dtype: object
```

7.14.2 `DataFrames`

- Enhanced two-dimensional `array`
- Can have custom row and column indices
- Offers additional operations and capabilities that make them more convenient for many data-science oriented tasks
- Support missing data
- Each column in a `DataFrame` is a `Series`

Creating a `DataFrame` from a Dictionary

- Create a `DataFrame` from a dictionary that represents student grades on three exams

In [1]:

```
import pandas as pd
```

In [2]:

```
grades_dict = {'Wally': [87, 96, 70], 'Eva': [100, 87, 90],
               'Sam': [94, 77, 90], 'Katie': [100, 81, 82],
               'Bob': [83, 65, 85]}
```

In [3]:

```
grades = pd.DataFrame(grades_dict)
```

- Pandas displays `DataFrame` s in tabular format with indices *left aligned* in the index column and the remaining columns' values *right aligned*

In [4]:

```
grades
```

Out[4]:

	Wally	Eva	Sam	Katie	Bob
0	87	100	94	100	83
1	96	87	77	81	65
2	70	90	90	82	85

Customizing a `DataFrame`'s Indices with the `index` Attribute

- Can use the **`index` attribute** to change the `DataFrame`'s indices from sequential integers to labels
- Must provide a one-dimensional collection that has the same number of elements as there are *rows* in the `DataFrame`

In [5]:

```
grades.index = ['Test1', 'Test2', 'Test3']
```

In [6]:

```
grades
```

Out[6]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

Accessing a `DataFrame` 's Columns

- Can quickly and conveniently look at your data in many different ways, including selecting portions of the data
- Get `Eva` 's grades by name
- Displays her column as a `Series`

In [7]:

```
grades['Eva']
```

Out[7]:

```
Test1    100
Test2     87
Test3     90
Name: Eva, dtype: int64
```

- If a `DataFrame` 's column-name strings are valid Python identifiers, you can use them as attributes

In [8]:

```
grades.Sam
```

Out[8]:

```
Test1    94
Test2    77
Test3    90
Name: Sam, dtype: int64
```

Selecting Rows via the `loc` and `iloc` Attributes

- `DataFrame` s support indexing capabilities with `[]` , but pandas documentation recommends using the attributes `loc` , `iloc` , `at` and `iat`
  - Optimized to access `DataFrame` s and also provide additional capabilities
- Access a row by its label via the `DataFrame` 's **`loc` attribute**

In [9]:

```
grades.loc['Test1']
```

Out[9]:

```
Wally     87
Eva      100
Sam       94
Katie    100
Bob       83
Name: Test1, dtype: int64
```

- Access rows by integer zero-based indices using the **`iloc` attribute** (the `i` in `iloc` means that it's used with integer indices)

In [10]:

```
grades.iloc[1]
```

Out[10]:

```
Wally    96
Eva      87
Sam      77
Katie    81
Bob      65
Name: Test2, dtype: int64
```

Selecting Rows via Slices and Lists with the `loc` and `iloc` Attributes

- Index can be a *slice*
- When using slices containing **labels** with `loc` , the range specified **includes** the high index (`'Test3'`):

In [11]:

```
grades.loc['Test1':'Test3']
```

Out[11]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65
Test3	70	90	90	82	85

- When using slices containing **integer indices** with `iloc`, the range you specify **excludes** the high index (2):

In [12]:

```
grades.iloc[0:2]
```

Out[12]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test2	96	87	77	81	65

- Select *specific rows* with a *list*

In [13]:

```
grades.loc[['Test1', 'Test3']]
```

Out[13]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test3	70	90	90	82	85

In [14]:

```
grades.iloc[[0, 2]]
```

Out[14]:

	Wally	Eva	Sam	Katie	Bob
Test1	87	100	94	100	83
Test3	70	90	90	82	85

Selecting Subsets of the Rows and Columns

- View only `Eva` 's and `Katie` 's grades on `Test1` and `Test2`

In [15]:

```
grades.loc['Test1':'Test2', ['Eva', 'Katie']]
```

Out[15]:

	Eva	Katie
Test1	100	100
Test2	87	81

- Use `iloc` with a list and a slice to select the first and third tests and the first three columns for those tests

In [16]:

```
grades.iloc[[0, 2], 0:3]
```

Out[16]:

	Wally	Eva	Sam
Test1	87	100	94
Test3	70	90	90

Boolean Indexing

- One of pandas' more powerful selection capabilities is **Boolean indexing**
- Select all the A grades—that is, those that are greater than or equal to 90:
  - Pandas checks every grade to determine whether its value is greater than or equal to 90 and, if so, includes it in the new `DataFrame`.
  - Grades for which the condition is `False` are represented as **`NaN`** **(not a number)** in the new `DataFrame
  - `NaN` is pandas' notation for missing values

In [17]:

```
grades[grades >= 90]
```

Out[17]:

	Wally	Eva	Sam	Katie	Bob
Test1	NaN	100.0	94.0	100.0	NaN
Test2	96.0	NaN	NaN	NaN	NaN
Test3	NaN	90.0	90.0	NaN	NaN

- Select all the B grades in the range 80–89

In [18]:

```
grades[(grades >= 80) & (grades < 90)]
```

Out[18]:

	Wally	Eva	Sam	Katie	Bob
Test1	87.0	NaN	NaN	NaN	83.0
Test2	NaN	87.0	NaN	81.0	NaN
Test3	NaN	NaN	NaN	82.0	85.0

- Pandas Boolean indices combine multiple conditions with the Python operator `&` (bitwise AND), *not* the `and` Boolean operator
- For `or` conditions, use `|` (bitwise OR)
- NumPy also supports Boolean indexing for `array` s, but always returns a one-dimensional array containing only the values that satisfy the condition

Accessing a Specific `DataFrame` Cell by Row and Column

- `DataFrame` method **`at`** and **`iat`** attributes get a single value from a `DataFrame`

In [19]:

```
grades.at['Test2', 'Eva']
```

Out[19]:

```
87
```

In [20]:

```
grades.iat[2, 0]
```

Out[20]:

```
70
```

- Can assign new values to specific elements

In [21]:

```
grades.at['Test2', 'Eva'] = 100
```

In [22]:

```
grades.at['Test2', 'Eva']
```

Out[22]:

```
100
```

In [23]:

```
grades.iat[1, 2] = 87
```

In [24]:

```
grades.iat[1, 2]
```

Out[24]:

87

Descriptive Statistics

- `DataFrame` s **`describe` method** calculates basic descriptive statistics for the data and returns them as a `DataFrame`
- Statistics are calculated by column

In [25]:

```
grades.describe()
```

Out[25]:

	Wally	Eva	Sam	Katie	Bob
count	3.000000	3.000000	3.000000	3.000000	3.000000
mean	84.333333	96.666667	90.333333	87.666667	77.666667
std	13.203535	5.773503	3.511885	10.692677	11.015141
min	70.000000	90.000000	87.000000	81.000000	65.000000
25%	78.500000	95.000000	88.500000	81.500000	74.000000
50%	87.000000	100.000000	90.000000	82.000000	83.000000
75%	91.500000	100.000000	92.000000	91.000000	84.000000
max	96.000000	100.000000	94.000000	100.000000	85.000000

- Quick way to summarize your data
- Nicely demonstrates the power of array-oriented programming with a clean, concise functional-style call
- Can control the precision and other default settings with pandas' **`set_option` function**

In [26]:

```
pd.set_option('precision', 2)
```

In [27]:

```
grades.describe()
```

Out[27]:

	Wally	Eva	Sam	Katie	Bob
count	3.00	3.00	3.00	3.00	3.00
mean	84.33	96.67	90.33	87.67	77.67
std	13.20	5.77	3.51	10.69	11.02
min	70.00	90.00	87.00	81.00	65.00
25%	78.50	95.00	88.50	81.50	74.00
50%	87.00	100.00	90.00	82.00	83.00
75%	91.50	100.00	92.00	91.00	84.00
max	96.00	100.00	94.00	100.00	85.00

- For student grades, the most important of these statistics is probably the mean
- Can calculate that for each student simply by calling `mean` on the `DataFrame`

In [28]:

```
grades.mean()
```

Out[28]:

```
Wally     84.33
Eva       96.67
Sam       90.33
Katie     87.67
Bob       77.67
dtype: float64
```

Transposing the `DataFrame` with the `T` Attribute

- Can quickly **transpose** rows and columns—so the rows become the columns, and the columns become the rows—by using the **`T` attribute** to get a view

In [29]:

```
grades.T
```

Out[29]:

	Test1	Test2	Test3
Wally	87	96	70
Eva	100	100	90
Sam	94	87	90
Katie	100	81	82
Bob	83	65	85

- Assume that rather than getting the summary statistics by student, you want to get them by test
- Call `describe` on `grades.T`

In [30]:

```
grades.T.describe()
```

Out[30]:

	Test1	Test2	Test3
count	5.00	5.00	5.00
mean	92.80	85.80	83.40
std	7.66	13.81	8.23
min	83.00	65.00	70.00
25%	87.00	81.00	82.00
50%	94.00	87.00	85.00
75%	100.00	96.00	90.00
max	100.00	100.00	90.00

- Get average of all the students' grades on each test

In [31]:

```
grades.T.mean()
```

Out[31]:

```
Test1    92.8
Test2    85.8
Test3    83.4
dtype: float64
```

Sorting by Rows by Their Indices

- Can sort a `DataFrame` by its rows or columns, based on their indices or values
- Sort the rows by their *indices* in *descending* order using **`sort_index`** and its keyword argument `ascending=False`

In [32]:

```
grades.sort_index(ascending=False)
```

Out[32]:

	Wally	Eva	Sam	Katie	Bob
Test3	70	90	90	82	85
Test2	96	100	87	81	65
Test1	87	100	94	100	83

Sorting by Column Indices

- Sort columns into ascending order (left-to-right) by their column names
- **`axis=1` keyword argument** indicates that we wish to sort the *column* indices, rather than the row indices
  - `axis=0` (the default) sorts the *row* indices

In [33]:

```
grades.sort_index(axis=1)
```

Out[33]:

	Bob	Eva	Katie	Sam	Wally
Test1	83	100	100	94	87
Test2	65	100	81	87	96
Test3	85	90	82	90	70

Sorting by Column Values

- To view `Test1` 's grades in descending order so we can see the students' names in highest-to-lowest grade order, call method **`sort_values`**
- `by` and `axis` arguments work together to determine which values will be sorted
  - In this case, we sort based on the column values (`axis=1`) for `Test1`

In [34]:

```
grades.sort_values(by='Test1', axis=1, ascending=False)
```

Out[34]:

	Eva	Katie	Sam	Wally	Bob
Test1	100	100	94	87	83
Test2	100	81	87	96	65
Test3	90	82	90	70	85

- Might be easier to read the grades and names if they were in a column
- Sort the transposed `DataFrame` instead

In [35]:

```
grades.T.sort_values(by='Test1', ascending=False)
```

Out[35]:

	Test1	Test2	Test3
Eva	100	100	90
Katie	100	81	82
Sam	94	87	90
Wally	87	96	70
Bob	83	65	85

- Since we're sorting only `Test1`'s grades, we might not want to see the other tests at all
- Combine selection with sorting

In [36]:

```
grades.loc['Test1'].sort_values(ascending=False)
```

Out[36]:

```
Katie     100
Eva       100
Sam        94
Wally      87
Bob        83
Name: Test1, dtype: int64
```

Copy vs. In-Place Sorting

- `sort_index` and `sort_values` return a *copy* of the original `DataFrame`
- Could require substantial memory in a big data application
- Can sort *in place* by passing the keyword argument `inplace=True`

7.17 Intro to Data Science: `pandas` Series and `DataFrames`

- NumPy's `array` is optimized for homogeneous numeric data that's accessed via integer indices
- Big data applications must support mixed data types, customized indexing, missing data, data that's not structured consistently and data that needs to be manipulated into forms appropriate for the databases and data analysis packages you use
- **Pandas** is the most popular library for dealing with such data
- Two key collections
    - **`Series`** for one-dimensional collections
    - **`DataFrames`** for two-dimensional collections
- NumPy and pandas are intimately related
    - `Series` and `DataFrame`s use `array`s "under the hood"
    - `Series` and `DataFrame`s are valid arguments to many NumPy operations
    - `array`s are valid arguments to many `Series` and `DataFrame` operations

---